

Universidad Simón Bolívar  
Departamento de Cómputo Científico y Estadística  
CO-5412 - Optimización No Lineal I

# **Máquinas de Aprendizaje con Mínimos Cuadrados no Lineales**

Rubmary Rojas 13-11264

Rafael Blanco 13-10156

Enero-Marzo 2018

# Contenido

<b>1</b>	<b>Introducción</b>	<b>2</b>
<b>2</b>	<b>Desarrollo teórico de optimización</b>	<b>3</b>
<b>3</b>	<b>Generación de los datos</b>	<b>6</b>
3.1	Generación con distribución uniforme . . . . .	6
3.2	Generación con dos nubes . . . . .	6
3.3	Preprocesamiento de datos . . . . .	7
<b>4</b>	<b>Resultados experimentales</b>	<b>8</b>
4.1	Datos generados con distribución uniforme, linealmente separables ( $e = 0$ ) . . . . .	8
4.2	Datos con ruido . . . . .	10
4.3	Datos en formas de dos nubes . . . . .	11
<b>5</b>	<b>Conclusiones</b>	<b>14</b>
<b>6</b>	<b>Anexos</b>	<b>15</b>

# 1 Introducción

Se quiere crear una máquina que aprenda a clasificar entre dos clases mediante un ajuste por mínimos cuadrados no lineales. El modelo propuesto es equivalente a una red neuronal artificial sin capas ocultas con una sola neurona en la capa de salida y una función de activación no lineal, esquemáticamente se puede representar de la siguiente manera:

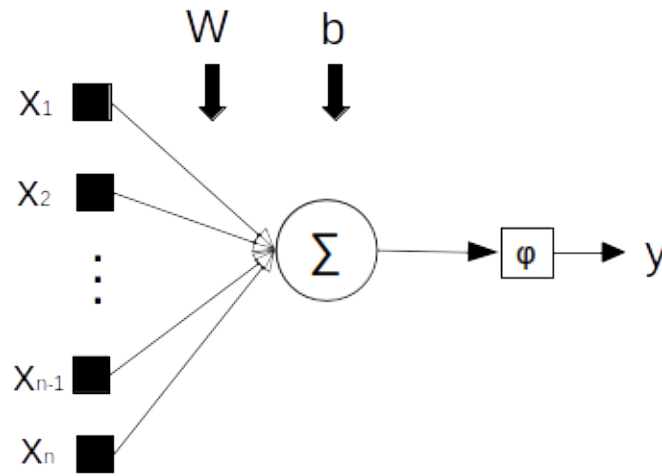


Figura 1: Red neuronal

Se utiliza como función de activación  $\varphi$ , la tangente hiperbólica, es decir  $\varphi(x) = \tanh(\rho x)$ , donde  $\rho$  es un parámetro a establecer.

Se realizará aprendizaje supervisado, pues se conoce la respuesta de los datos de entrenamiento, y se quiere minimizar la siguiente función, en la cual  $N$  representa la cantidad de datos de entrenamiento y  $x_k \in \mathbb{R}^n$ , son los datos de entrada que tienen dimensión  $n$ :

$$ET(\vec{w}, b) = \frac{1}{2} \sum_{k=1}^N e_k^2$$

donde  $e_k = y_k - d_k$ ,  $y_k = \varphi(V_k)$  y  $V_k = \vec{w}^t \vec{x}_k + b = \sum_{j=1}^n w_j x_{kj} + b$ .

## 2 Desarrollo teórico de optimización

Para encontrar el mínimo de la función  $ET$ , se aplicarán las condiciones de optimalidad, sin embargo, es conveniente rescribir la ecuación anterior, considerando  $\vec{w} \in \mathbb{R}^{(m+1)}$ , asignando  $w_0 = b$ , y  $x_0 = 1$ , de esta forma asumimos que ya no se tiene sesgo y la función se puede expresar como:

$$\begin{aligned} ET(\vec{w}) &= \frac{1}{2} \sum_{i=1}^N e_k^2 = \frac{1}{2} \sum_{i=1}^N (y_k - d_k)^2 \\ &= \frac{1}{2} \sum_{i=1}^N (\varphi(\vec{w}^t \vec{x}_k) - d_k)^2 \\ &= \frac{1}{2} \sum_{i=1}^N \left( \varphi \left( \sum_{j=0}^n w_j x_{kj} \right) - d_k \right)^2 \end{aligned}$$

Calculemos ahora  $\frac{\partial ET}{\partial w_i}$ , usando la linealidad de la derivación y aplicando regla de la cadena varias veces, se obtiene que:

$$\frac{\partial ET}{\partial w_i} = \frac{1}{2} \sum_{k=1}^N \frac{\partial e_k^2}{\partial w_i} = \frac{1}{2} \sum_{k=1}^N \frac{\partial e_k^2}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{V_k} \frac{\partial V_k}{\partial w_i} \quad (***)$$

Por otra parte, se tiene que:

$$\begin{aligned} \frac{\partial e_k^2}{\partial e_k} &= 2e_k = 2(y_k - d_k) \\ \frac{\partial e_k}{\partial y_k} &= 1 \\ \frac{\partial y_k}{V_k} &= \varphi'(V_k) \\ \frac{\partial V_k}{\partial w_i} &= \sum_{j=0}^n \frac{\partial (w_j x_{kj})}{\partial w_i} = x_{ki} \end{aligned}$$

Sustituyendo las expresiones anteriores en (\*\*\*), resulta:

$$\begin{aligned}
\frac{\partial ET}{\partial w_i} &= \frac{1}{2} \sum_{k=1}^N \frac{\partial e_k^2}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{V_k} \frac{\partial V_k}{\partial w_i} \\
&= \frac{1}{2} \sum_{k=1}^N 2(y_k - d_k)(1)\varphi'(V_k)x_{ki} \\
&= \sum_{k=1}^N (y_k - d_k)\varphi'(V_k)x_{ki} \\
&= \sum_{k=1}^N (\varphi(V_k) - d_k)\varphi'(V_k)x_{ki}
\end{aligned}$$

Sea  $X \in \mathbb{R}^N \times \mathbb{R}^{n+1}$  la matriz donde la  $i$ -ésima fila es el vector  $\vec{x}_i$  y sea  $\phi \in \mathbb{R}^N$ , tal que  $\phi_k = (\varphi(V_k) - d_k)\varphi'(V_k)$ , entonces se puede escribir el gradiente como:

$$\nabla ET = X^t \phi$$

En este caso en particular, la función de activación  $\varphi$  es la tangente hiperbólica, luego

$$\begin{aligned}
\tanh'(x) &= \left( \frac{\sinh(x)}{\cosh(x)} \right)' \\
&= \frac{\sinh'(x)\cosh(x) - \cosh'(x)\sinh(x)}{\cosh^2(x)} \\
&= \frac{\cosh^2(x) - \sinh^2(x)}{\cosh^2(x)} \\
&= \frac{1}{\cosh^2 x}
\end{aligned}$$

Obteniendo que:

$$\varphi'(V_k) = \rho \cdot \tanh'(\rho V_k) = \frac{\rho}{\cosh^2(\rho V_k)}$$

Ahora calculemos las derivadas parciales de segundo orden y el hessiano de la función

$$\begin{aligned}
\frac{\partial ET}{\partial w_j \partial w_i} &= \frac{\partial}{\partial w_j} \left( \sum_{k=1}^N (y_k - d_k) \varphi'(V_k) x_{ki} \right) \\
&= \sum_{k=1}^N \frac{\partial((y_k - d_k) \varphi'(V_k) x_{ki})}{\partial w_j} \\
&= \sum_{k=1}^N x_{ki} \frac{\partial((y_k - d_k) \varphi'(V_k))}{\partial w_j} \\
&= \sum_{k=1}^N x_{ki} \left( \frac{\partial(y_k - d_k)}{\partial w_j} \varphi'(V_k) + \frac{\partial \varphi'(V_k)}{\partial w_j} (y_k - d_k) \right) \\
&= \sum_{k=1}^N x_{ki} \left( \frac{\partial \varphi(V_k)}{\partial w_j} \varphi'(V_k) + \frac{\partial \varphi'(V_k)}{\partial w_j} (y_k - d_k) \right)
\end{aligned}$$

Por otra parte, se tiene que:

$$\begin{aligned}
\frac{\partial \varphi(V_k)}{\partial w_j} &= \frac{\partial \varphi(V_k)}{\partial V_k} \frac{\partial V_k}{\partial w_j} = \varphi'(V_k) x_{kj} \\
\frac{\partial \varphi'(V_k)}{\partial w_j} &= \frac{\partial \varphi'(V_k)}{\partial V_k} \frac{\partial V_k}{\partial w_j} = \varphi''(V_k) x_{kj}
\end{aligned}$$

Luego, se tiene que:

$$\begin{aligned}
\frac{\partial ET}{\partial w_j \partial w_i} &= \sum_{k=1}^N x_{ki} \left( \frac{\partial \varphi(V_k)}{\partial w_j} \varphi'(V_k) + \frac{\partial \varphi'(V_k)}{\partial w_j} (y_k - d_k) \right) \\
&= \sum_{k=1}^N x_{ki} (\varphi'(V_k) x_{kj} \varphi'(V_k) + \varphi''(V_k) x_{kj} (y_k - d_k)) \\
&= \sum_{k=1}^N x_{ki} x_{kj} (\varphi'(V_k)^2 + \varphi''(V_k) (y_k - d_k))
\end{aligned}$$

Por último, es necesario calcular  $\varphi''(x)$ :

$$\begin{aligned}
\varphi''(x) &= \left( \frac{\rho}{\cosh^2(\rho x)} \right)' = \rho(1 - \varphi^2(x))' \\
&= \rho(-2\varphi(x))(\varphi'(x)) = -2\rho\varphi(x)\varphi'(x)
\end{aligned}$$

### 3 Generación de los datos

Se generaron datos aleatorios en 2 y 3 dimensiones. Separar datos en  $2D$  es equivalente a encontrar una recta que divida al plano en dos semiplano, cada uno con puntos de una sólo categoría y separar datos en  $3D$  es equivalente a encontrar un plano. Se generaron datos de dos formas diferentes:

#### 3.1 Generación con distribución uniforme

En este generador se utilizan dos parámetros que corresponden a la entrada del programa, el primero es la distancia, que se denotará con  $d$  y el segundo, denotado con  $e$  el máximo ruido que se agrega a los datos. En este caso se genera de forma aleatoria los parámetros  $A$ ,  $B$  y  $C$  de la recta  $Ax + By + C = 0$ , con distribución uniforme en el intervalo  $(-3, 3)$  (agregando el parámetro  $D$  en el caso que se trabaje en 3 dimensiones). Una vez con la recta establecida, se crean puntos, cuyas coordenadas son generadas con distribución uniforme en el intervalo  $(-10, 10)$ .

Una vez se tiene el dato, este es descartado si  $abs(Ax + By + C) < d$ , y luego es clasificado según el signo del producto anterior, luego a cada componente se le agrega un error entre  $(0, e)$  de forma aleatoria. De esta forma se pueden generar datos con una franja entre ellos con una distancia dada, o incluso datos que se mezclen en su frontera, según los parámetros que se le proporcionen al generador.

#### 3.2 Generación con dos nubes

En este caso se utilizan los parámetros  $d$  y  $var$ . Primero se eligen los centros de forma aleatoria con una distancia  $d$  entre ellos. Luego para cada centro, los datos son elegidos seleccionando primero la distancia que tendrán al centro con una distribución normal y varianza  $var$ , descartándose aquellos valores que estén a una distancia mayor que 1.5 para evitar que la data tenga demasiados datos atípicos.

### 3.3 Preprocesamiento de datos

Al intentar ejecutar los algoritmos con los datos obtenidos inicialmente, se observó que en la mayoría de los casos no se podía obtener un resultado, siendo la respuesta dada por el programa para cada peso *nan*. Por lo que se trasladaron y rescalaron los datos, para que las componentes tuvieran una media igual a 0 y una varianza igual a 1, una vez hecho esto fue posible aplicar los algoritmos.



## 4 Resultados experimentales

Para resolver el problema de optimización se utilizaron dos algoritmos diferentes de búsqueda lineal. Primero se utilizó el método de cauchy, en el cual se utiliza el un vector con dirección opuesta al gradiente como dirección de búsqueda. Como segundo algoritmo se utilizó un método Casi Newton, BFGS, en el cual se hacen aproximaciones al hessiano de la función.

En ambos métodos se emplea el algoritmo de backtracking para definir la longitud del paso. Como condición de parada se verificó que el gradiente tuviera una norma menor que  $10^{-3}$  o que se alcanzaran un total de 10000 iteraciones.

Los datos probados, con sus resultados fueron los siguientes. La tabla muestra el valor del parámetro  $\rho$ , el número de llamadas a la función,  $\#f$ , el número de llamadas al gradiente de la función  $\#\nabla f$ , la norma del gradiente final, el número de iteraciones  $\#it$ , la cantidad de datos correctos (c), la cantidad de datos mal clasificados (w), el porcentaje de precisión (p) y el tiempo empleado en encontrar la solución (t):

### 4.1 Datos generados con distribución uniforme, linealmente separables ( $e = 0$ )

Los siguientes resultados se obtuvieron al experimentar con datos generados de la primera forma (distribución uniforme), y sin ruido, por lo que la data es linealmente separable. En todos los casos se obtuvo una clasificación perfecta por parte de ambos algoritmos y se puede notar que bfgs converge con mayor rapidez que cauchy. Los resultados obtenidos cuando  $\rho = 1$  (de la función objetivo) se presenta en la siguiente tabla (en el tipo *Sep* indica que los datos se generaron con una separación entre ellos):

tipo	método	# $f$	# $\nabla f$	$ \nabla f $	#it	c	w	p	t
2D	cauchy	326	135	0.0009858	134	100	0	100%	0.043145
2D	bfgs	41	29	3.12E-05	14	100	0	100%	0.006597
3D	cauchy	195	78	0.000996067	77	100	0	100%	0.026426
3D	bfgs	111	33	1.99E-05	16	100	0	100%	0.015444
Sep 2D	cauchy	1928	694	0.000999768	693	100	0	100%	0.252256
Sep 2D	bfgs	13	9	6.96E-05	4	100	0	100%	0.002086
Sep 3D	cauchy	929	229	0.000996687	228	100	0	100%	0.117985
Sep 3D	bfgs	9	5	0.000359558	2	100	0	100%	0.001427

Tabla 1: Datos linealmente separables

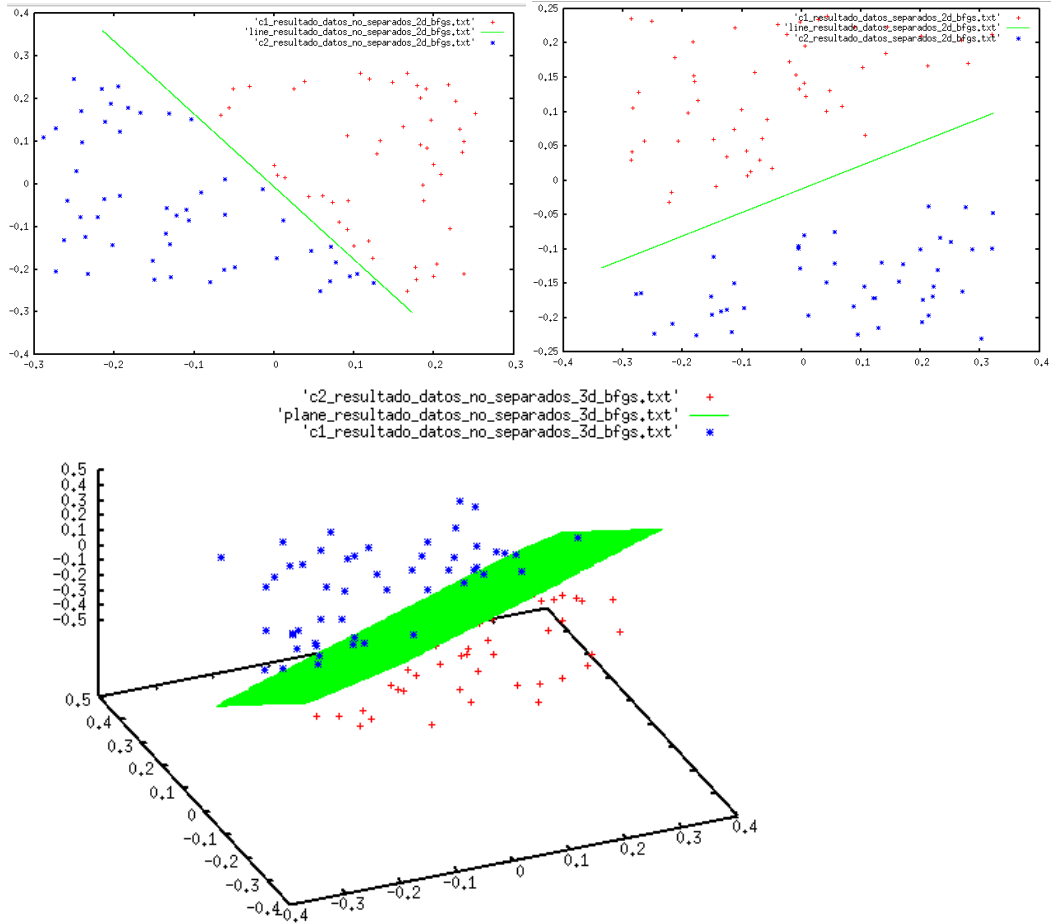


Figura 2: Resultados de datos linealmente separables usando bfgs

## 4.2 Datos con ruido

En este caso la data no resulta linealmente separable, por lo que la precisión no puede ser del 100%, no obstante se genera un clasificador con una precisión mayor del 90% en todos los casos. En cuanto a precisión se puede observar que el método bfgs fue un poco más preciso al clasificar los datos en 3D, por otra parte se sigue observando como predomina el método casi newton en cuanto a rapidez de convergencia. La siguiente tabla muestra los resultados para el valor de  $\rho$  más eficiente:

datos	método	$\rho$	$\#f$	$\#\nabla f$	$ \nabla f $	$\#it$	c	w	p	t
2D	Cauchy	0.6	6966	1567	0.000998582	1566	96	4	96%	0.86841
2D	bfgs	0.5	47	29	3.58E-05	14	96	4	96%	0.007323
3D	Cauchy	0.9	12748	1774	0.00098584	1773	96	4	96%	1.54784
3D	bfgs	0.8	59	39	1.65E-07	19	98	2	98%	0.009909

Tabla 2: Datos con ruido

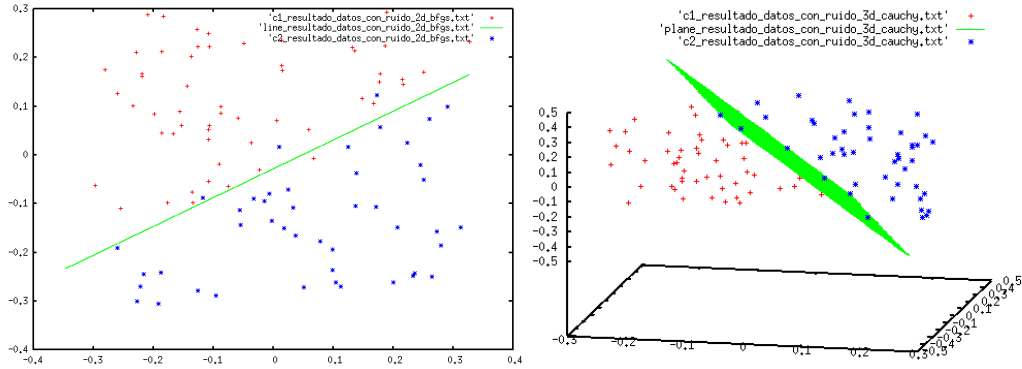


Figura 3: Datos con ruido

### 4.3 Datos en formas de dos nubes

En las tablas presentadas en esta sección se puede observar que en este caso se logra una separación perfecta de los datos en todos los casos, también se observa que la cantidad de iteraciones es menor a medida que se utiliza un valor de  $\rho$  mayor. Además, también se puede observar que hay una diferencia significativa en la cantidad de iteraciones entre los métodos, siendo BFGS mucho más rápido. También se experimentó con datos en 3 dimensiones y se obtienen conclusiones similares, no obstante con los datos generados se obtuvo un dato mal clasificado.

$\rho$	$\#f$	$\#\nabla f$	$ \nabla f $	$\#it$	c	w	p	t
0.1	9422	4685	0.000999834	4684	100	0	100%	1.33706
0.2	5217	2535	0.00099976	2534	100	0	100%	0.717469
0.3	3768	1789	0.000999999	1788	100	0	100%	0.514781
0.4	2818	1353	0.000999527	1352	100	0	100%	0.385423
0.5	1935	962	0.000999841	961	100	0	100%	0.267034
0.6	619	310	0.000999309	309	100	0	100%	0.085609
0.7	33	17	0.000925588	16	100	0	100%	0.004632
0.8	47	24	0.000995901	23	100	0	100%	0.006499
0.9	3	2	0.000644333	1	100	0	100%	0.000458
1	3	2	8.24E-05	1	100	0	100%	0.000457

Tabla 3: Método de Cauchy

$\rho$	$\#f$	$\#\nabla f$	$ \nabla f $	$\#it$	c	w	p	t
0.1	41	29	6.0656e-05	14	100	0	100%	0.018883
0.2	32	23	4.64513e-05	11	100	0	100%	0.006914
0.3	49	33	0.000108441	16	100	0	100%	0.008335
0.4	39	27	0.00093442	13	100	0	100%	0.006406
0.5	32	23	7.27122e-07	11	100	0	100%	0.005167
0.6	18	13	0.000432188	6	100	0	100%	0.00293
0.7	19	13	0.000487316	6	100	0	100%	0.003491
0.8	25	17	0.000327908	8	100	0	100%	0.003951
0.9	18	13	9.41778e-05	6	100	0	100%	0.002914
1	16	11	0.000161781	5	100	0	100%	0.002535

Tabla 4: Método BFGS

En este tipo de data, se observó un caso que se puede observar que es linealmente separable, ambos algoritmos clasifican bien para valores pequeños de  $\rho$ , no obstante, para el método de cauchy se pierde precisión cuando  $\rho$  disminuye. Otro punto a destacar es que en algunos caso el algoritmo bfgs no puede resolver el problema debido a asuntos de overflow y el resultado del programa es *nan*, en este caso la precisión es de 0%. Esto se puede observar, tanto en la data de 2 dimensiones, como en la data de 3 dimensiones. La siguiente tabla muestra la precisión de ambos algoritmos según el valor de  $\rho$  utilizado:

$\rho$	cauchy 3D	bfgs 2D	cauchy 3D	bfgs 3D
0.1	97.14%	98.57%	100%	100%
0.2	97.14%	98.57%	100%	100%
0.3	97.14%	98.57%	100%	100%
0.4	98.57%	0%	100%	100%
0.5	98.57%	0%	100%	100%
0.6	98.57%	98.57%	100%	100%
0.7	98.57%	98.57%	99%	0%
0.8	92.86%	0%	85%	100%
0.9	91.43%	0%	85%	100%
1	91.43%	98.57%	85%	100%

Tabla 5: Comparación de precisión

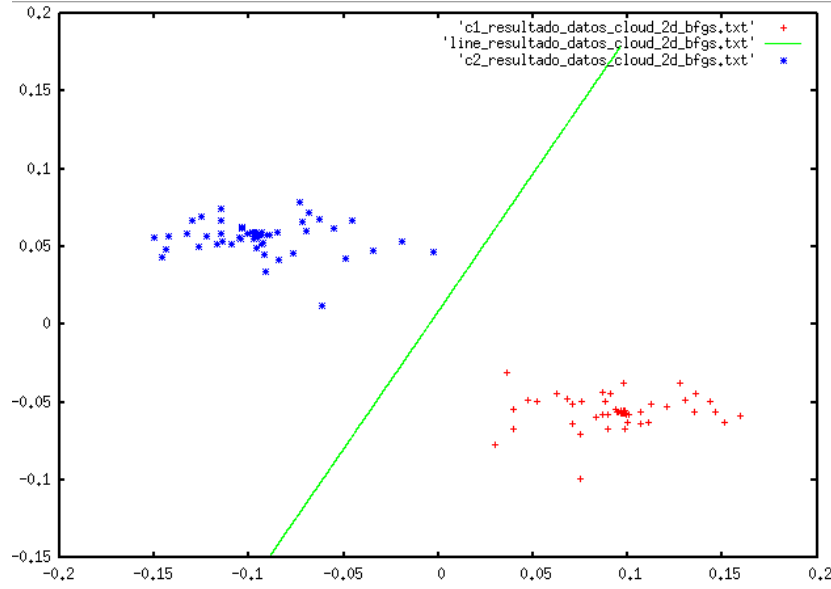


Figura 4: Nubes en 2d

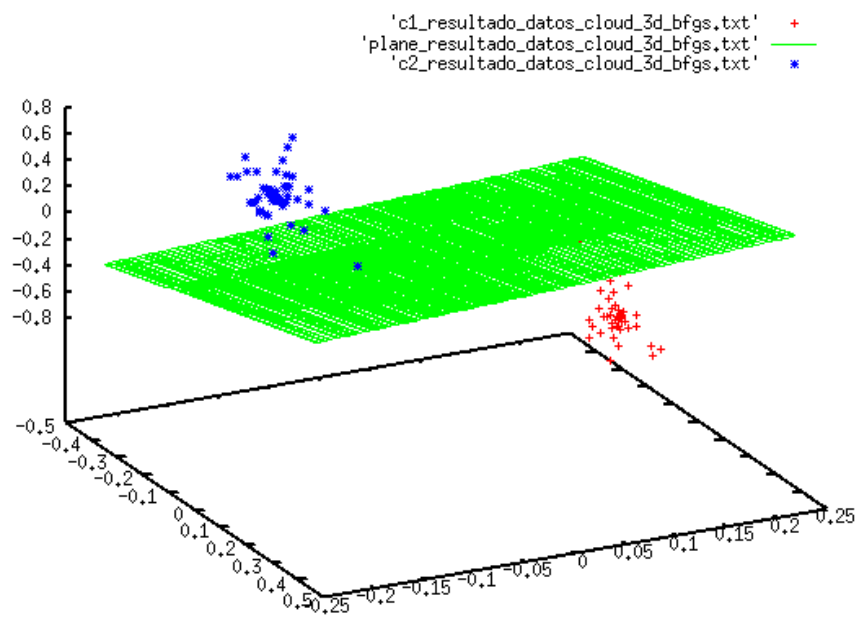


Figura 5: Nubes en 3d

## 5 Conclusiones

Luego de realizar los experimentos con los distintos casos de prueba se puede concluir que tanto el algoritmo de Cauchy, como el algoritmo de BFGS convergen con soluciones adecuadas para este problema de optimización no lineal. Se puede concluir además que el desempeño del algoritmo BFGS es mucho mejor que el del algoritmo de Cauchy, ya que la cantidad de iteraciones necesarias para converger es mucho menor, de hecho, generalmente no pasa de 20 iteraciones, disminuyendo notoriamente el número de evaluaciones de la función objetivo, del gradiente y, por ende, el tiempo de cómputo de este método.

Otra observación importante que se debe destacar es que para datos con ruido, es decir, aquellos datos que no son linealmente separables, se obtuvo una mejor clasificación cuando el rho de la función objetivo toma valores centrales en el intervalo entre 0 y 1.

## 6 Anexos

Los algoritmos fueron implementados en el lenguaje de programación C++, todo el código utilizado es incluido en la siguiente sección:

### Funciones de vectores y matrices

```
1  #include <vector>
2  #include <cmath>
3  #define sq(X) (X)*(X)
4  #define cub(X) (X)*(X)*(X)
5  typedef std::vector<std::vector<double> > matrix;
6  typedef std::vector<double> vector;
7
8  matrix cross(vector A, vector B)
9  {
10     int n = A.size(), m = B.size();
11     matrix C = std::vector<vector>(n, vector(m, 0));
12     for (int i = 0; i < n; i++)
13     {
14         for (int j = 0; j < m; j++)
15             C[i][j] = A[i]*B[j];
16     }
17     return C;
18 }
19 matrix operator + (matrix A, matrix B)
20 {
21     int n = A.size(), m = A[0].size();
22     matrix C(n, vector(m));
23     for (int i = 0; i < n; i++)
24     {
25         for (int j = 0; j < m; j++)
26             C[i][j] = A[i][j] + B[i][j];
27     }
28     return C;
29 }
30 vector operator * (matrix A, vector B)
31 {
32     int n = A.size(), m = B.size();
33     vector C(n, 0);
34 }
```



```

35     for (int i = 0; i < n; i++)
36     {
37         for (int j = 0; j < m; j++)
38             C[i] += A[i][j]*B[j];
39     }
40
41     return C;
42 }
43 vector operator + (vector A, vector B)
44 {
45     int n = A.size();
46     vector C(n);
47     for (int i = 0; i < n; i++)
48         C[i] = A[i] + B[i];
49     return C;
50 }
51 vector operator * (double eta, vector A)
52 {
53     int n = A.size();
54     vector ans(n);
55     for (int i = 0; i < n; i++)
56         ans[i] = eta*A[i];
57     return ans;
58 }
59 vector operator - (vector A, vector B)
60 {
61     int n = A.size();
62     vector C(n);
63     for (int i = 0; i < n; i++)
64         C[i] = A[i] - B[i];
65     return C;
66 }
67 double operator * (vector A, vector B)
68 {
69     int n = A.size();
70     double ans = 0;
71     for (int i = 0; i < n; i++)
72         ans += A[i]*B[i];
73     return ans;
74 }
75

```

```

76 matrix operator * (matrix A, matrix B)
77 {
78     int n = A.size(), m = B.size(), t = B[0].size();
79     matrix C(n, vector(t));
80     for (int i = 0; i < n; i++){
81         for (int j = 0; j < t; j++) {
82             C[i][j] = 0;
83             for (int k = 0; k < m; k++)
84                 C[i][j] += A[i][k]*B[k][j];
85         }
86     }

```

## Implementación de la clase *Function* y la clase *ET*

```
1 #include "vector_functions.cpp"
2 #include <iostream>
3
4 /**
5  * Class Fuction:
6  * clase abstracta para representar las funciones que
7  * seran pasadas como parametro a la busqueda lineal
8  * @N:          dimension del vector de entrada
9  * @total:      numero de llamadas a la funcion
10 * @total_d:    numero de llamadas a la derivada de la funcion
11 * @total_h:    numero de llamadas al hessiando de la funcion
12 * @total_i:    numero de veces que se calcula la inversa de
                  una matriz
13 */
14 class Function {
15 public:
16     int N;
17     int total;
18     int total_d;
19     int total_h;
20     int total_i;
21     /**
22      * val - valor de la funcion evualuada en un punto
23      * Parametros:
24      * x - vector en el que se evalua la funcion
25      * Descripcion:
26      * retorna el valor de la funcion,
27      * por defecto es la funcion constate
28      * (igual a 1)
29      */
30     virtual double val(vector x){
31         return 1;
32     }
33
34     /**
35      * d - derivada
36      * Parametros:
37      * x - vector, punto a evaluar la derivada
38      * Descripcion:
```

```

39     * retorna el valor de la derivada en el
40     * punto dado
41     */
42     virtual vector d(vector x) {
43         return vector(0);
44     }
45     /**
46     * h - hessiano
47     * Parametros:
48     * x - vector, punto a evaluar el hessiano
49     * retorna el valor del hessiano en el
50     * punto dado
51     */
52     virtual matrix h(vector x) {
53         return matrix(0);
54     }
55     /**
56     * inverse - inversa
57     * Parametros:
58     * A - matriz de entrada
59     * Descripcion
60     * retorna la inversa de la
61     * matriz dada como parametro
62     */
63     virtual matrix inverse(matrix &A){
64         return matrix(0);
65     }
66 };
67
68 class ET : public Function{
69 public:
70     int M;
71     matrix X;
72     vector V, PHI, y, D;
73     double rho;
74
75     ET(int NO, matrix X, vector D, double rho = 1) :
76         X(X), D(D), rho(rho) {
77         N = NO;
78         M = X.size();
79         y = vector(M);

```

```

80     PHI = vector(M);
81 }
82 double phi(double x) {
83     x *= rho;
84     return (exp(x) - exp(-x))/(exp(x) + exp(-x));
85 }
86 double dphi(double x) {
87     x *= rho;
88     return rho*sq(2/(exp(x) + exp(-x)));
89 }
90 double ddphi(double x) {
91     return -2*rho*phi(x)*dphi(x);
92 }
93 void precalculations(vector w) {
94     V = X*w;
95     for (int k = 0; k < M; k++)
96         y[k] = phi(V[k]);
97     for (int k = 0; k < M; k++)
98         PHI[k] = (y[k] - D[k])*dphi(V[k]);
99 }
100 virtual double val(vector w) {
101     total++;
102     precalculations(w);
103     double e = 0;
104     for (int k = 0; k < M; k++)
105         e += sq(y[k]-D[k]);
106     return e/2;
107 }
108 virtual vector d(vector w) {
109     total_d++;
110     matrix Xt = t(X);
111     return Xt*PHI;
112 }
113 virtual matrix h(vector w) {
114     total_h++;
115     matrix H(N, vector(N));
116     for (int i = 0; i < N; i++) {
117         for (int j = i; j < N; j++){
118             H[i][j] = 0;
119             for (int k = 0; k < M; k++)
120                 H[i][j] += X[k][i]*X[k][j]*(sq(dphi(V[k])))

```

```

121         + ddphi(V[k])*(y[k]-D[k]));
122         H[j][i] = H[i][j];
123     }
124     return H;
125 }
126 double det_adj(int i, int j, matrix &M) {
127     matrix A(2, vector(2));
128     int i2 = 0, j2 = 0;
129     for (int i1 = 0; i1 < N; i1++) {
130         j2 = 0;
131         if (i1 == i) continue;
132         for (int j1 = 0; j1 < N; j1++){
133             if (j1 == j) continue;
134             A[i2][j2] = M[i1][j1];
135             j2++;
136         }
137         i2++;
138     }
139     return A[0][0]*A[1][1]-A[1][0]*A[0][1];
140 }
141 virtual matrix inverse(matrix &M) {
142     total_i++;
143     matrix X = matrix(N, vector(N));
144     double det = M[0][0]*(M[1][1]*M[2][2] - M[1][2]*M
145                 [2][1])
146                 - M[0][1]*(M[1][0]*M[2][2] - M[1][2]*M
147                 [2][0])
148                 + M[0][2]*(M[1][0]*M[2][1] - M[1][1]*M
149                 [2][0]);
150     matrix Mt = t(M);
151     for (int i = 0; i < N; i++){
152         for (int j = 0; j < N; j++){
153             X[i][j] = det_adj(i, j, Mt)*(((i+j)&1) ? -1 :
154             1);
155             X[i][j] /= det;
156         }
157     }
158     return X;
159 }
160 };

```

## Implementación de la clase *LinearSearch*, *Cauchy* y *Bfgs*

```
1 #include "Function.cpp"
2 #include <climits>
3 #include <cstdlib>
4 #include <ctime>
5
6 /**
7  * Class LinearSearch:
8  * clase abstracta para representar las diferentes
9  * búsquedas lineales
10 * @eps:   precision utilizada como condicion de parada
11 * @MAX_IT: numero maximo de iteraciones permitidas
12 * @f:     funcion utilizada para optimizar
13 * @x:     valor del punto en cada iteracion
14 * @gx:    valor del gradiente en cada iteracion
15 * @fx:    valor de la funcion en cada iteracion
16 */
17 class LinearSearch {
18 public:
19     double eps;
20     int MAX_IT;
21     Function *f;
22     vector x;
23     vector gx;
24     double fx;
25
26     /**
27      * alpha - longitud del paso
28      * Parametros:
29      * @d - vector de la direccion de busqueda
30      * Descripcion:
31      * Longitud del paso en el algoritmo iterativo,
32      * su valor por defecto es 1
33      */
34     virtual double alpha(vector &d)
35     {
36         return 1;
37     }
38
39     /**
```

```

40     * dir - direccion de busqueda
41     * Descripcion:
42     * Retorna la direccion de descenso
43     * del algoritmo, por defecto es la
44     * direccion opuesta al gradiente
45     */
46     virtual vector dir()
47     {
48         return -1*gx;
49     }
50
51     /**
52     * linear_search
53     * Parametros:
54     * @x0 - vector inicial
55     * Descripcion:
56     * Algoritmo de busqueda lineal,
57     * converge a la solucion que es
58     * almacenada en la variable x
59     */
60     int linear_search(vector x0)
61     {
62         vector d;
63         x = x0;
64         int k = 0;
65         while (true) {
66             k++;
67             fx = f -> val(x);
68             gx = f -> d(x);
69             if (abs(gx) < eps || k > MAX_IT)
70                 break;
71             d = dir();
72             x = x + alpha(d)*d;
73         }
74         return k-1;
75     }
76 };
77
78 /**
79 * Class Cauchy
80 * Subclase hija de LinearSearch

```



```

81  * @al0:    paso inicial en el algoritmo de backtraking
82  * gamma:   parametro del algoritmo de backtracking
83  * @ro:     parametro del algoritmo de backtraking
84  * @sig:     parametro que puede ser randomizado para
85  *           variaciones de Cauchy, por defecto es igual a 1
86  */
87  class Cauchy : public LinearSearch
88  {
89  public:
90      double al0, gamma, ro, sig;
91
92      /**
93       * Cauchy- Constructor de la clase
94       * Descripcion:
95       * Establece los parametros para el metodo de
96       * cauchy
97       */
98      Cauchy(double al0, double gamma, double ro, double sig = 1)
99          :
100            al0(al0), gamma(gamma), ro(ro), sig(sig) { }
101
102      /**
103       * Sigma -
104       * Descripcion:
105       * retorna el valor del parametro sigma
106       */
107      virtual double sigma()
108      {
109          return sig;
110      }
111
112      /**
113       * alpha - longitud del paso
114       * Descripcion:
115       * sobrescribe la funcion de la clase padre
116       * utiliza el algoritmo de backtraking para
117       * para calcular la longitud del paso
118       */
119      virtual double alpha(vector &d)
120      {
121          double C = gamma*(gx*d), a = al0;

```

```

121         while(f -> val(x + a*d) > fx + a*C){
122             a = ro*a*sigma();
123         }
124         return a;
125     }
126 };
127
128 /**
129  * Class Cauchy
130  * Subclase hija de Cauchy
131  */
132 class Cauchy2 : public Cauchy {
133 public:
134
135     /**
136      * Cauchy2- Constructor de la clase
137      * Descripcion:
138      * Establece los parametros para el metodo de
139      * cauchy e inicializa la semilla para generar
140      * los numeros aleatorios
141      */
142     Cauchy2(double al0, double gamma, double ro = 0.5, double
        sig = 1) :
143         Cauchy(al0, gamma, ro, sig) {
144         srand(time(NULL));
145     }
146
147     /**
148      * sigma
149      * Descripcion:
150      * sobrescribe la funcion de la clase padre
151      * utiliza el algoritmo de backtraking para
152      * para calcular la longitud del paso
153      */
154     virtual double sigma()
155     {
156         return (double) std::rand()/INT_MAX;
157     }
158 };
159
160

```

```

161  /**
162   * Class Newton
163   * Subclase hija de Linear Search
164   */
165  class Newton : public LinearSearch
166  {
167      /**
168       * dir - direccion de busqueda
169       * Descripcion:
170       * sobrescribe la funcion de la clase padre
171       * calcular la direccion de busqueda
172       */
173      virtual vector dir() {
174          matrix H = f -> h(x);
175          H = f -> inverse(H);
176          return -1*(H*gx);
177      }
178  };
179
180  class Bfgs : public Cauchy
181  {
182  public:

```

### *Implementación de los generadores*

```
1 #include <iostream>
2 #include <climits>
3 #include <cstdlib>
4 #include <ctime>
5 #include <algorithm>
6 using namespace std;
7
8 double get_rand(double a, double b){
9     double r = (double) std::rand()/INT_MAX;
10    return r*(b - a) + a;
11 }
12
13 int main() {
14     double A, B, C;
15     srand(time(NULL));
16     A = get_rand(-3, 3);
17     B = get_rand(-3, 3);
18     C = get_rand(-3, 3);
19     int N;
20     double distance, error;
21
22     cin >> N >> distance >> error;
23     cout << N << endl;
24     int i = 0;
25     while(i < N) {
26         double X, Y;
27         X = get_rand(-10, 10);
28         Y = get_rand(-10, 10);
29
30         if (abs(A*X + B*Y + C) < distance)
31             continue;
32         cout << X + get_rand(-error/2, error/2) << '␣' << Y +
33             get_rand(-error/2, error/2) << '␣';
34         if (A*X + B*Y + C > 0)
35             cout << 1;
36         else
37             cout << -1;
38         cout << endl;
39         i++;
40     }
```

39		}
40		}

```

1  #include <iostream>
2  #include <climits>
3  #include <cstdlib>
4  #include <ctime>
5  #include <algorithm>
6  using namespace std;
7
8  double get_rand(double a, double b){
9      double r = (double) std::rand()/INT_MAX;
10     return r*(b - a) + a;
11 }
12
13 int main() {
14     double A, B, C, D;
15     srand(time(NULL));
16     A = get_rand(-3, 3);
17     B = get_rand(-3, 3);
18     C = get_rand(-3, 3);
19     D = get_rand(-3, 3);
20
21     int N;
22     double distance, error;
23     cin >> N >> distance >> error;
24     cout << N << endl;
25
26     int i = 0;
27     while (i < N) {
28         double X, Y, Z;
29         X = get_rand(-10, 10);
30         Y = get_rand(-10, 10);
31         Z = get_rand(-10, 10);
32
33         if (abs(A*X + B*Y + C*Z + D) < distance)
34             continue;
35
36         cout << X + get_rand(-error/2, error/2) << '␣' << Y+
37             get_rand(-error/2, error/2) << '␣' << Z+get_rand(-error
38                 /2, error/2) << '␣';
39         if (A*X + B*Y + C*Z + D > 0)
40             cout << 1;
41         else

```

```
40         cout << -1;
41     cout << endl;
42     i++;
43 }
44 }
```

```

1  #include <iostream>
2  #include <climits>
3  #include <cstdlib>
4  #include <ctime>
5  #include <algorithm>
6  using namespace std;
7
8  double get_rand(double a, double b){
9      double r = (double) std::rand()/INT_MAX;
10     return r*(b - a) + a;
11 }
12
13 vector <double> rand_vector(double r) {
14     double x = get_rand(-1, 1), y = get_rand(-1, 1);
15     double norm = sqrt(x*x + y*y);
16     return vector <double>({x*r/norm, y*r/norm});
17 }
18
19 int main() {
20
21     srand(time(NULL));
22     int N;
23     double distance, var;
24     cin >> N >> distance >> var;
25     default_random_engine generator;
26     normal_distribution<double> distribution(0, var);
27     vector <double> c1, c2;
28     c1 = rand_vector(1);
29     c2 = rand_vector(distance);
30     c2[0] += c1[0];
31     c2[1] += c1[1];
32
33     cout << N << endl;
34
35     for (int i = 0; i < N; i++){
36         double X1, Y1, X2, Y2, r1, r2;
37         vector <double> d1, d2;
38         r1 = distribution(generator);
39         r2 = distribution(generator);
40         while (r1 > 1.5*var)
41             r1 = distribution(generator);

```



```

42     while (r2 > 1.5*var)
43         r2 = distribution(generator);
44
45     d1 = rand_vector(r1);
46     d2 = rand_vector(r2);
47
48     X1 = c1[0] + d1[0];
49     Y1 = c1[1] + d1[1];
50     X2 = c2[0] + d2[0];
51     Y2 = c2[1] + d2[1];
52
53     cout << X1 << ' ' << Y1 << ' ' << 1 << endl;
54     cout << X2 << ' ' << Y2 << ' ' << -1 << endl;
55 }
56 }

```

```

1  #include <iostream>
2  #include <climits>
3  #include <cstdlib>
4  #include <ctime>
5  #include <algorithm>
6  using namespace std;
7
8  double get_rand(double a, double b){
9      double r = (double) std::rand()/INT_MAX;
10     return r*(b - a) + a;
11 }
12
13 vector <double> rand_vector(double r) {
14     double x = get_rand(-1, 1), y = get_rand(-1, 1), z = get_rand
        (-1, 1);
15     double norm = sqrt(x*x + y*y + z*z);
16     return vector <double>({x*r/norm, y*r/norm, z*r/norm});
17 }
18
19 int main() {
20
21     srand(time(NULL));
22     int N;
23     double distance, var;
24     cin >> N >> distance >> var;
25     default_random_engine generator;
26     normal_distribution<double> distribution(0, var);
27     vector <double> c1, c2;
28     c1 = rand_vector(1);
29     c2 = rand_vector(distance);
30
31     for (int i = 0; i < 3; i++)
32         c2[i] += c1[i];
33
34     cout << N << endl;
35
36     for (int i = 0; i < N; i++){
37         double X1, Y1, Z1, X2, Y2, Z2, r1, r2;
38
39         vector <double> d1, d2;
40         r1 = distribution(generator);

```

```

41     r2 = distribution(generator);
42     while (r1 > 1.5*var)
43         r1 = distribution(generator);
44     while (r2 > 1.5*var)
45         r2 = distribution(generator);
46
47     d1 = rand_vector(r1);
48     d2 = rand_vector(r2);
49
50     X1 = c1[0] + d1[0];
51     Y1 = c1[1] + d1[1];
52     Z1 = c1[2] + d1[2];
53
54     X2 = c2[0] + d2[0];
55     Y2 = c2[1] + d2[1];
56     Z2 = c2[2] + d2[2];
57
58     cout << X1 << ' ' << Y1 << ' ' << Z1 << ' ' << 1 << endl;
59     cout << X2 << ' ' << Y2 << ' ' << Z2 << ' ' << -1 << endl;
60 }
61 }

```

*Implementació de las funciones main que llaman a los algoritmos*

```
1 #include "LinearSearch.cpp"
2 #include <iostream>
3 #include <fstream>
4 #include <ctime>
5 #include <climits>
6 #include <cstdlib>
7 #include <algorithm>
8 #define EPS 1e-3
9
10 void make_data(matrix &X, vector &D){
11     vector x(3, 1);
12     int M;
13     std::cin >> M;
14     D = vector(M);
15     for (int i = 0; i < M; i++) {
16         std::cin >> x[1] >> x[2] >> D[i];
17         X.push_back(x);
18     }
19     double miuX = 0, varX = 0, miuY = 0, varY = 0;
20     for (int i = 0; i < M; i++) {
21         miuX += X[i][1];
22         miuY += X[i][2];
23     }
24     miuX /= M;
25     miuY /= M;
26     for (int i = 0; i < M; i++) {
27         varX += sq(X[i][1] - miuX);
28         varY += sq(X[i][2] - miuY);
29     }
30     varX /= (M-1);
31     varY /= (M-1);
32     for (int i = 0; i < M; i++) {
33         X[i][1] = (X[i][1] - miuX)/varX;
34         X[i][2] = (X[i][2] - miuY)/varY;
35     }
36 }
37 void make_graphics(vector w, matrix X, vector D){
38     double C = w[0], B = w[2], A = w[1];
39     std::ofstream line("line.txt"), c1("c1.txt"), c2("c2.txt");
```

```

40  int N = X.size();
41  double minY = 1e100, maxY = 1e-100, minX = 1e100, maxX = 1e
    -100;
42
43  for (int i = 0; i < N; i++){
44      minX = std::min(minX, X[i][1]);
45      maxX = std::max(maxX, X[i][1]);
46      minY = std::min(minY, X[i][2]);
47      maxY = std::max(maxY, X[i][2]);
48  }
49  minX -= 0.05;
50  maxY += 0.05;
51  minY -= 0.05;
52  maxY += 0.05;
53  if (std::abs(A) > EPS){
54      B /= A;
55      C /= A;
56      A = 1;
57  }
58  if (std::abs(B) < EPS) {
59      line << -C/A << '□' << minY << std::endl;
60      line << -C/A << '□' << maxY << std::endl;
61  }else {
62      int total = 10000;
63      for (int i = 0; i < total; i++){
64          double x0 = i*(maxX - minX)/total + minX, y0 = -(C + x0)/
              B;
65          if (minY <= y0 && y0 <= maxY)
66              line << x0 << '□' << y0 << std::endl;
67      }
68  }
69  for (int i = 0; i < N; i++) {
70      if(D[i] == 1)
71          c1 << X[i][1] << '□' << X[i][2] << std::endl;
72      else
73          c2 << X[i][1] << '□' << X[i][2] << std::endl;
74  }
75  }
76  std::vector<int> results(vector w, matrix X, vector D) {
77      int correct = 0, wrong = 0;
78      int M = X.size();

```

```

79     for (int i = 0; i < M; i++){
80         double cross = w*X[i];
81         if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
82             correct++;
83         else
84             wrong++;
85     }
86     return std::vector<int>({correct, wrong});
87 }
88
89 double get_rand(double a, double b){
90     double r = (double) std::rand()/INT_MAX;
91     return r*(b - a) - a;
92 }
93
94 int main() {
95     matrix X;
96     vector D;
97     make_data(X, D);
98     //ET *f = new ET(3, X, D, 0.5);
99     vector w0(3, 0);
100    srand(time(NULL));
101    int k;
102    clock_t t1, t2;
103    double ro = 0.3;
104    Cauchy *cauchy = new Cauchy(10, 1e-4, ro);
105    cauchy -> eps = EPS;
106    cauchy -> MAX_IT = 1000000;
107    std::cout << "Valor_de_rho_rho_f_objetivo_#
        llamadas_f_objetivo_#llamadas_gradiente_
        Norma_del_gradiente_";
108    std::cout << "#iteraciones_Datos_bien_clasificados_
        Datos_mal_clasificados_Precisin_Tiempo_de_computo\n";
109    for (double i=0.1; i<=1; i=i+0.1){
110        ET *f = new ET(3, X, D, i);
111        cauchy -> f = f;
112        t1 = clock();
113        k = cauchy -> linear_search(w0);
114        t2 = clock();
115        vector w = cauchy -> x;
116        std::vector<int> r = results(w, X, D);

```

```

117     make_graphics(w, X, D);
118     std::cout << ro << "␣" << i << "␣" << f->total << "␣" << f
        -> total_d << "␣" << std::abs(cauchy -> gx) << "␣";
119     std::cout << k << "␣" << r[0] << "␣" << r[1] << "␣" <<
        100.00*r[0]/(r[0]+r[1]) << "%" << "␣" << (double) (t2 -
        t1)/CLOCKS_PER_SEC;
120     std::cout << std::endl;
121     delete f;
122 }
123 }

```

```

1  #include "LinearSearch.cpp"
2  #include <iostream>
3  #include <fstream>
4  #include <ctime>
5  #include <climits>
6  #include <cstdlib>
7  #include <algorithm>
8  #define EPS 1e-3
9
10 int N;
11 void make_data(matrix &X, vector &D){
12     vector x(4, 1);
13     int M;
14     std::cin >> M;
15     D = vector(M);
16     for (int i = 0; i < M; i++) {
17         std::cin >> x[1] >> x[2] >> x[3];
18         std::cin >> D[i];
19         X.push_back(x);
20     }
21     double miuX = 0, varX = 0;
22     double miuY = 0, varY = 0;
23     double miuZ = 0, varZ = 0;
24     for (int i = 0; i < M; i++) {
25         miuX += X[i][1];
26         miuY += X[i][2];
27         miuZ += X[i][3];
28     }
29     miuX /= M;
30     miuY /= M;
31     miuZ /= M;
32     for (int i = 0; i < M; i++) {
33         varX += sq(X[i][1] - miuX);
34         varY += sq(X[i][2] - miuY);
35         varZ += sq(X[i][3] - miuZ);
36     }
37     varX /= (M-1);
38     varY /= (M-1);
39     varZ /= (M-1);
40
41     for (int i = 0; i < M; i++) {

```



```

42     X[i][1] = (X[i][1] - miuX)/varX;
43     X[i][2] = (X[i][2] - miuY)/varY;
44     X[i][3] = (X[i][3] - miuZ)/varZ;
45 }
46 }
47
48 void make_graphics(vector w, matrix X, vector D){
49     double D1 = w[0], A = w[1], B = w[2], C = w[3];
50     std::ofstream plane("plane.txt"), c1("c1.txt"), c2("c2.txt");
51     int N = X.size();
52
53     double minX = 1e100, maxX = -1e100, minY = 1e100, maxY = 1e
        -100, minZ = 1e100, maxZ = 1e-100;
54
55     for (int i = 0; i < N; i++){
56         minX = std::min(minX, X[i][1]);
57         maxX = std::max(maxX, X[i][1]);
58         minY = std::min(minY, X[i][2]);
59         maxY = std::max(maxY, X[i][2]);
60         minZ = std::min(minZ, X[i][3]);
61         maxZ = std::max(maxZ, X[i][3]);
62     }
63
64     minX -= 0.1;
65     minY -= 0.1;
66     minZ -= 0.1;
67     maxX += 0.1;
68     maxY += 0.1;
69     maxZ += 0.1;
70
71     int total = 50;
72     for (int i = 1; i <= total; i++) {
73         for (int j = 1; j <= total; j++) {
74             double x0 = i*(maxX - minX)/total + minX;
75             double y0 = j*(maxY - minY)/total + minY;
76             double z0 = -(D1 + A*x0 + B*y0)/C;
77             if (minZ <= z0 && z0 <= maxZ)
78                 plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
79         }
80     }
81     for (int j = 1; j <= total; j++) {

```

```

82     for (int i = 1; i <= total; i++) {
83         double x0 = i*(maxX - minX)/total + minX;
84         double y0 = j*(maxY - minY)/total + minY;
85         double z0 = -(D1 + A*x0 + B*y0)/C;
86         if (minZ <= z0 && z0 <=maxZ)
87             plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
88     }
89 }
90 for (int i = 0; i < N; i++) {
91     if(D[i] == 1)
92         c1 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std
93             ::endl;
94     else
95         c2 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std
96             ::endl;
97 }
98 std::vector<int> results(vector w, matrix X, vector D) {
99     int correct = 0, wrong = 0;
100    int M = X.size();
101    for (int i = 0; i < M; i++){
102        double cross = w*X[i];
103        if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
104            correct++;
105        else
106            wrong++;
107    }
108    return std::vector<int>({correct, wrong});
109 }
110 double get_rand(double a, double b){
111     double r = (double) std::rand()/INT_MAX;
112     return r*(b - a) - a;
113 }
114
115 int main() {
116     matrix X;
117     vector D;
118     make_data(X, D);
119     ET *f = new ET(4, X, D, 0.5);
120     vector w0(4, 0);

```

```

121 srand(time(NULL));
122 int k;
123 clock_t t1, t2;
124 double ro = 0.5;
125 Cauchy *cauchy = new Cauchy(10, 1e-4, ro);
126 cauchy -> eps = EPS;
127 cauchy -> MAX_IT = 100000;
128 cauchy -> f = f;
129 t1 = clock();
130 k = cauchy -> linear_search(w0);
131 t2 = clock();
132 vector w = cauchy -> x;
133 std::vector<int> r = results(w, X, D);
134 make_graphics(w, X, D);
135 std::cout << "Valor_de_rho:_" << ro << std::endl;
136 std::cout << "Total_de_llamadas_a:_" << std::endl;
137 std::cout << "\tFuncion:_" << f -> total << std::endl;
138 std::cout << "\tDerivadas:_" << f -> total_d << std::endl;
139 std::cout << "\tHessiano:_" << f -> total_h << std::endl;
140 std::cout << "\tInversa:_" << f -> total_i << std::endl;
141 std::cout << "Norma_del_gradiente:_" << abs(cauchy -> gx) <<
    std::endl;
142 std::cout << "Total_de_iteraciones:_" << k << std::endl;
143 std::cout << "Resultados:_" << std::endl;
144 std::cout << "\tDatos_bien_clasificados:_" << r[0] << std::
    endl;
145 std::cout << "\tDatos_mal_clasificados:_" << r[1] << std::
    endl;
146 std::cout << "\tPrecision:_" << 100.00*r[0]/(r
    [0]+r[1]) << "%" << std::endl;
147 std::cout << "Tiempo_utilizado:_" << (double) (t2 - t1)/
    CLOCKS_PER_SEC << std::endl;
148 }

```

```

1  #include "LinearSearch.cpp"
2  #include <iostream>
3  #include <fstream>
4  #include <ctime>
5  #include <climits>
6  #include <cstdlib>
7  #include <algorithm>
8  #define EPS 1e-3
9
10 void make_data(matrix &X, vector &D){
11     vector x(3, 1);
12     int M;
13     std::cin >> M;
14     D = vector(M);
15     for (int i = 0; i < M; i++) {
16         std::cin >> x[1] >> x[2] >> D[i];
17         X.push_back(x);
18     }
19     double miuX = 0, varX = 0, miuY = 0, varY = 0;
20     for (int i = 0; i < M; i++) {
21         miuX += X[i][1];
22         miuY += X[i][2];
23     }
24     miuX /= M;
25     miuY /= M;
26     for (int i = 0; i < M; i++) {
27         varX += sq(X[i][1] - miuX);
28         varY += sq(X[i][2] - miuY);
29     }
30     varX /= (M-1);
31     varY /= (M-1);
32     for (int i = 0; i < M; i++) {
33         X[i][1] = (X[i][1] - miuX)/varX;
34         X[i][2] = (X[i][2] - miuY)/varY;
35     }
36 }
37 void make_graphics(vector w, matrix X, vector D){
38     double C = w[0], B = w[2], A = w[1];
39     std::ofstream line("line.txt"), c1("c1.txt"), c2("c2.txt");
40     int N = X.size();
41     double minY = 1e100, maxY = 1e-100, minX = 1e100, maxX = 1e

```

```

-100;
42
43 for (int i = 0; i < N; i++){
44     minX = std::min(minX, X[i][1]);
45     maxX = std::max(maxX, X[i][1]);
46     minY = std::min(minY, X[i][2]);
47     maxY = std::max(maxY, X[i][2]);
48 }
49
50 minX -= 0.05;
51 maxY += 0.05;
52 minY -= 0.05;
53 maxY += 0.05;
54 if (std::abs(A) > EPS){
55     B /= A;
56     C /= A;
57     A = 1;
58 }
59
60 if (std::abs(B) < EPS) {
61     line << -C/A << '□' << minY << std::endl;
62     line << -C/A << '□' << maxY << std::endl;
63 }else {
64     int total = 10000;
65     for (int i = 0; i < total; i++){
66         double x0 = i*(maxX - minX)/total + minX, y0 = -(C + x0)/
            B;
67         if (minY <= y0 && y0 <= maxY)
68             line << x0 << '□' << y0 << std::endl;
69     }
70 }
71 for (int i = 0; i < N; i++) {
72     if(D[i] == 1)
73         c1 << X[i][1] << '□' << X[i][2] << std::endl;
74     else
75         c2 << X[i][1] << '□' << X[i][2] << std::endl;
76 }
77 }
78 std::vector<int> results(vector w, matrix X, vector D) {
79     int correct = 0, wrong = 0;
80     int M = X.size();

```

```

81     for (int i = 0; i < M; i++){
82         double cross = w*X[i];
83         if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
84             correct++;
85         else
86             wrong++;
87     }
88     return std::vector<int>({correct, wrong});
89 }
90
91 double get_rand(double a, double b){
92     double r = (double) std::rand()/INT_MAX;
93     return r*(b - a) - a;
94 }
95
96 matrix random_matrix(int n){
97     matrix id(n, vector(n));
98     for (int i=0; i<n; i++){
99         for (int j=0; j<n; j++){
100             id[i][j]=get_rand(0, 5);
101         }
102     }
103     id=id + t(id);
104     id= id + n*identity(n);
105     return id;
106 }
107
108 int main() {
109     matrix X, I;
110     vector D;
111     make_data(X, D);
112
113     ET *f = new ET(3, X, D, 0.5);
114     vector w0(3, 2);
115     srand(time(NULL));
116     for (int i = 0; i < w0.size(); i++)
117         w0[i] = get_rand(-1, 1);
118     int k;
119     clock_t t1, t2;
120     double ro = 0.3;
121

```

```

122 I=identity(3);
123 Bfgs *bfgs = new Bfgs(10, 1e-4, ro, I);
124 bfgs -> eps=EPS;
125 bfgs -> MAX_IT = 10000;
126 bfgs -> f = f;
127 t1 = clock();
128 k = bfgs -> linear_search(w0);
129 t2 = clock();
130 vector w = bfgs -> x;
131 std::vector<int> r = results(w, X, D);
132 make_graphics(w, X, D);
133 std::cout << "Valor_de_rho:_\n" << ro << std::endl;
134 std::cout << "Total_de_llamadas_a:_\n" << std::endl;
135 std::cout << "\tFuncion:_\n" << f -> total << std::endl;
136 std::cout << "\tDerivadas:_\n" << f -> total_d << std::endl;
137 std::cout << "\tHessiano:_\n" << f -> total_h << std::endl;
138 std::cout << "\tInversa:_\n" << f -> total_i << std::endl;
139 std::cout << "Norma_del_gradiente:_\n" << std::abs(bfgs -> gx)
    << std::endl;
140 std::cout << "Total_de_iteraciones:_\n" << k << std::endl;
141 std::cout << "Resultados:_\n" << std::endl;
142 std::cout << "\tDatos_bien_clasificados:_\n" << r[0] << std::
    endl;
143 std::cout << "\tDatos_mal_clasificados:_\n" << r[1] << std::
    endl;
144 std::cout << "\tPrecision:_\n" << 100.00*r[0]/(r
    [0]+r[1]) << "%" << std::endl;
145 std::cout << "Tiempo_utilizado:_\n" << (double) (t2 - t1)/
    CLOCKS_PER_SEC << std::endl;
146 }

```

```

1  #include "LinearSearch.cpp"
2  #include <iostream>
3  #include <fstream>
4  #include <ctime>
5  #include <climits>
6  #include <cstdlib>
7  #include <algorithm>
8  #define EPS 1e-3
9
10 int N;
11 void make_data(matrix &X, vector &D){
12     vector x(4, 1);
13     int M;
14     std::cin >> M;
15     D = vector(M);
16     for (int i = 0; i < M; i++) {
17         std::cin >> x[1] >> x[2] >> x[3];
18         std::cin >> D[i];
19         X.push_back(x);
20     }
21     double miuX = 0, varX = 0;
22     double miuY = 0, varY = 0;
23     double miuZ = 0, varZ = 0;
24     for (int i = 0; i < M; i++) {
25         miuX += X[i][1];
26         miuY += X[i][2];
27         miuZ += X[i][3];
28     }
29     miuX /= M;
30     miuY /= M;
31     miuZ /= M;
32     for (int i = 0; i < M; i++) {
33         varX += sq(X[i][1] - miuX);
34         varY += sq(X[i][2] - miuY);
35         varZ += sq(X[i][3] - miuZ);
36     }
37     varX /= (M-1);
38     varY /= (M-1);
39     varZ /= (M-1);
40     for (int i = 0; i < M; i++) {
41         X[i][1] = (X[i][1] - miuX)/varX;

```



```

42     X[i][2] = (X[i][2] - miuY)/varY;
43     X[i][3] = (X[i][3] - miuZ)/varZ;
44 }
45 }
46 void make_graphics(vector w, matrix X, vector D){
47     double D1 = w[0], A = w[1], B = w[2], C = w[3];
48     std::ofstream plane("plane.txt"), c1("c1.txt"), c2("c2.txt");
49     int N = X.size();
50     double minX = 1e100, maxX = -1e100, minY = 1e100, maxY = 1e
        -100, minZ = 1e100, maxZ = 1e-100;
51     for (int i = 0; i < N; i++){
52         minX = std::min(minX, X[i][1]);
53         maxX = std::max(maxX, X[i][1]);
54         minY = std::min(minY, X[i][2]);
55         maxY = std::max(maxY, X[i][2]);
56         minZ = std::min(minZ, X[i][3]);
57         maxZ = std::max(maxZ, X[i][3]);
58     }
59     minX -= 0.1;
60     minY -= 0.1;
61     minZ -= 0.1;
62     maxX += 0.1;
63     maxY += 0.1;
64     maxZ += 0.1;
65
66     int total = 50;
67     for (int i = 1; i <= total; i++) {
68         for (int j = 1; j <= total; j++) {
69             double x0 = i*(maxX - minX)/total + minX;
70             double y0 = j*(maxY - minY)/total + minY;
71             double z0 = -(D1 + A*x0 + B*y0)/C;
72             if (minZ <= z0 && z0 <= maxZ)
73                 plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
74         }
75     }
76
77     for (int j = 1; j <= total; j++) {
78         for (int i = 1; i <= total; i++) {
79             double x0 = i*(maxX - minX)/total + minX;
80             double y0 = j*(maxY - minY)/total + minY;
81             double z0 = -(D1 + A*x0 + B*y0)/C;

```

```

82         if (minZ <= z0 && z0 <=maxZ)
83             plane << x0 << '␣' << y0 << '␣' << z0 << std::endl;
84     }
85 }
86
87 for (int i = 0; i < N; i++) {
88     if(D[i] == 1)
89         c1 << X[i][1] << '␣' << X[i][2] << '␣' << X[i][3] << std
::endl;
90     else
91         c2 << X[i][1] << '␣' << X[i][2] << '␣' << X[i][3] << std
::endl;
92 }
93 }
94
95 std::vector<int> results(vector w, matrix X, vector D) {
96     int correct = 0, wrong = 0;
97     int M = X.size();
98     for (int i = 0; i < M; i++){
99         double cross = w*X[i];
100         if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
101             correct++;
102         else
103             wrong++;
104     }
105     return std::vector<int>({correct, wrong});
106 }
107 double get_rand(double a, double b){
108     double r = (double) std::rand()/INT_MAX;
109     return r*(b - a) - a;
110 }
111
112 int main() {
113     matrix X, I;
114     vector D;
115     make_data(X, D);
116     ET *f = new ET(4, X, D, 0.5);
117     vector w0(4, 0);
118     srand(time(NULL));
119     int k;
120     clock_t t1, t2;

```

```

121     double ro = 0.5;
122     I=identity(4);
123     Bfgs *bfgs = new Bfgs(10, 1e-4, ro, I);
124     bfgs -> eps=EPS;
125     bfgs -> MAX_IT = 10000;
126     bfgs -> f = f;
127     t1 = clock();
128     k = bfgs -> linear_search(w0);
129     t2 = clock();
130     vector w = bfgs -> x;
131     std::vector<int> r = results(w, X, D);
132     make_graphics(w, X, D);
133     std::cout << "Valor_de_rho:_" << ro << std::endl;
134     std::cout << "Total_de_llamadas_a:_" << std::endl;
135     std::cout << "\tFuncion:_" << f -> total << std::endl;
136     std::cout << "\tDerivadas:_" << f -> total_d << std::endl;
137     std::cout << "\tHessiano:_" << f -> total_h << std::endl;
138     std::cout << "\tInversa:_" << f -> total_i << std::endl;
139     std::cout << "Norma_del_gradiente:_" << abs(bfgs -> gx) <<
        std::endl;
140     std::cout << "Total_de_iteraciones:_" << k << std::endl;
141     std::cout << "Resultados:_" << std::endl;
142     std::cout << "\tDatos_bien_clasificados:_" << r[0] << std:::
        endl;
143     std::cout << "\tDatos_mal_clasificados:_" << r[1] << std:::
        endl;
144     std::cout << "\tPrecision:_" << 100.00*r[0]/(r
        [0]+r[1]) << "%" << std::endl;
145     std::cout << "Tiempo_utilizado:_" << (double) (t2 - t1)/
        CLOCKS_PER_SEC << std::endl;
146 }

1 CC = gcc
2 CXX = g++
3 C11 = -std=c++11
4 OPT = -Wall -O3 -Wno-unused-function -Wno-unused-variable
5
6 cauchy: src/cauchy.cpp
7     $(CXX) $(C11) $< -o $@
8
9 cauchy3D: src/cauchy3D.cpp

```

```

10     $(CXX) $(C11) $< -o $@
11
12 bfgs: src/bfgs.cpp
13     $(CXX) $(C11) $< -o $@
14
15 bfgs3D: src/bfgs3D.cpp
16     $(CXX) $(C11) $< -o $@
17
18 generator: src/generator.cpp
19     $(CXX) $(C11) $< -o $@
20
21 generator3D: src/generator3D.cpp
22     $(CXX) $(C11) $< -o $@
23
24 generator_clouds: src/generator_clouds.cpp
25     $(CXX) $(C11) $< -o $@
26
27 generator_clouds3D: src/generator_clouds3D.cpp
28     $(CXX) $(C11) $< -o $@
29
30
31 .PHONY: clean
32 clean:
33     rm -fr cauchy cauchy3D bfgs bfgs3D generador generator
34         generator3D generator_clouds generator_clouds3D
35
36 cleanData:
37     rm -fr *.txt

```