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

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 sóla neurona en la capa de salida y una función de activación no lineal, esquemáticamente se puede representar de la siguiente manera:

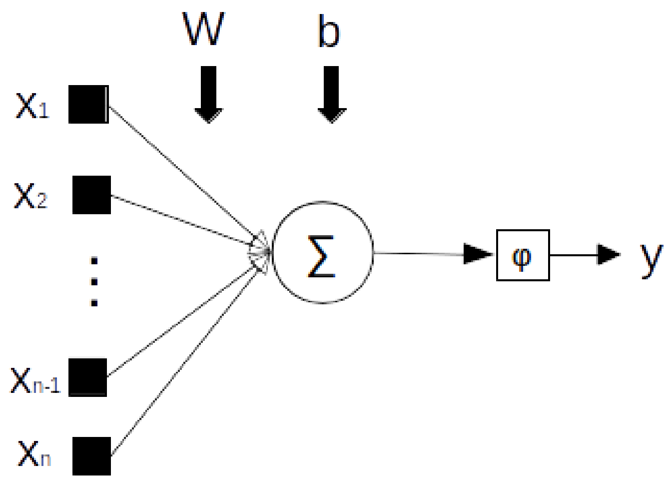

Figura 1: Red neuronal

Se utiliza como función de activación φ, la tangente hiperbólica, es decir $\varphi(x) = tanh(\rho x)$, dode ρ 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 dimesión 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 $V_k = \vec{w}^t \vec{x}_k + b = \sum_{j=1}^{n} w_i x_{ki} + b$.

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:

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

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 φ es la tangente hiperbólica, luego

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

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:

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

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 sóla 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 varizanza 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 alcazaran un total de 10000 iteraciones.

Los datos probados, con sus resultados fueron los siguientes. La tabla muestra el valor del parámetro ρ, 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 encotrar 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

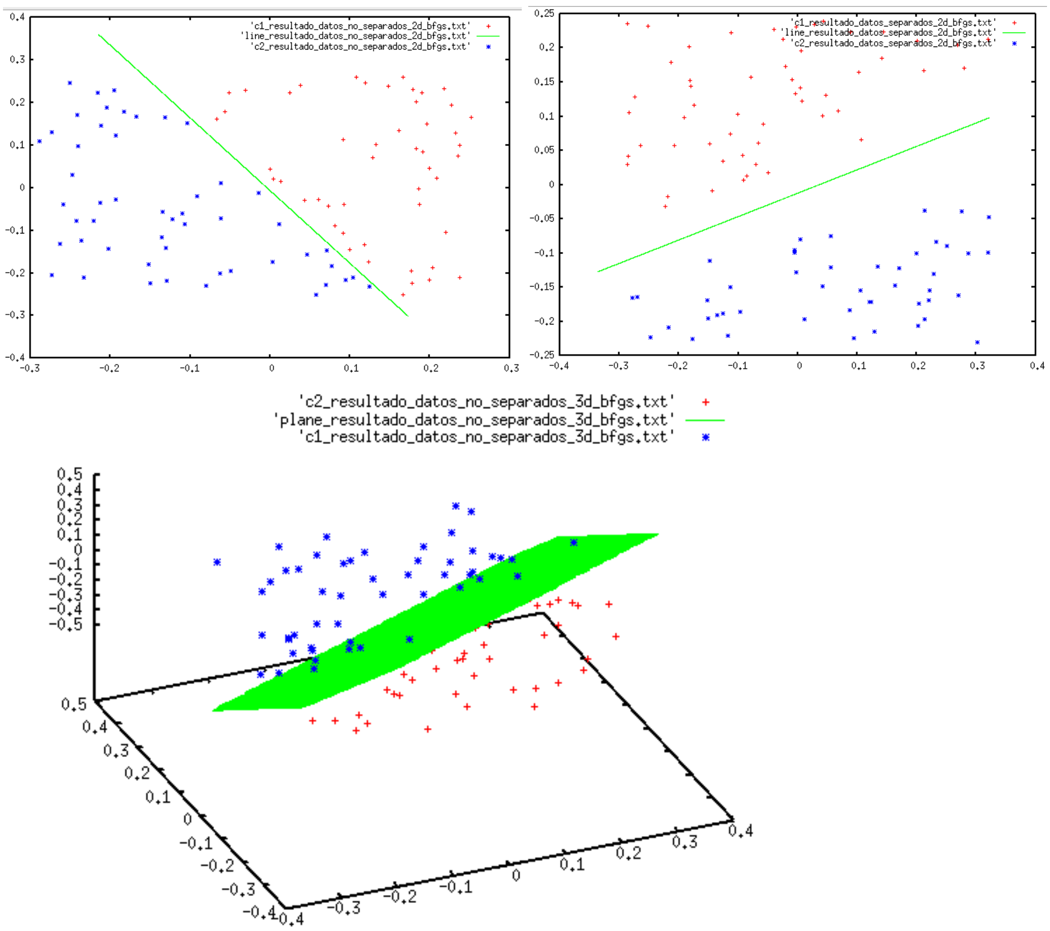

Figura 2: Resultados de datos linealmente separables usando bfgs

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 ρ más eficiente:

datos	método	ρ	$\#f$	$\#\nabla f$	$\|\nabla f\|$	#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

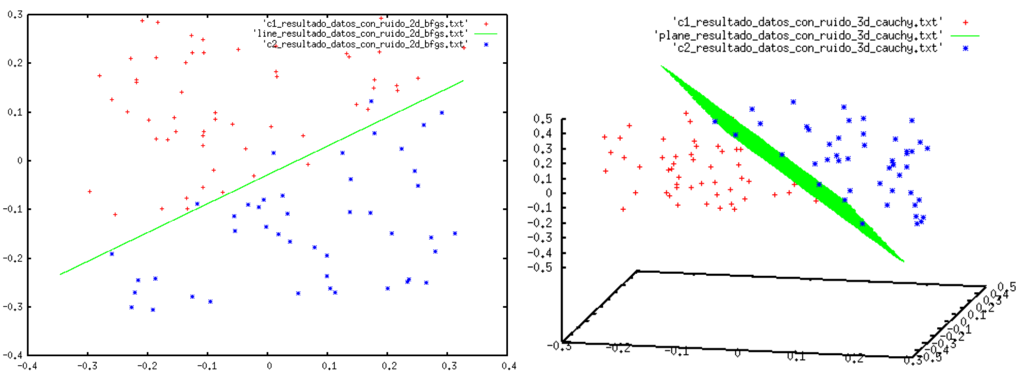

Figura 3: Datos con ruido

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 ρ 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.

ρ	$\#f$	$\#\nabla f$	$\|\nabla f\|$	#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

ρ	$\#f$	$\#\nabla f$	$\|\nabla f\|$	#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 ρ, no obstante, para el método de cauchy se pierde precisión cuando ρ 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 ρ utilizado:

ρ	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

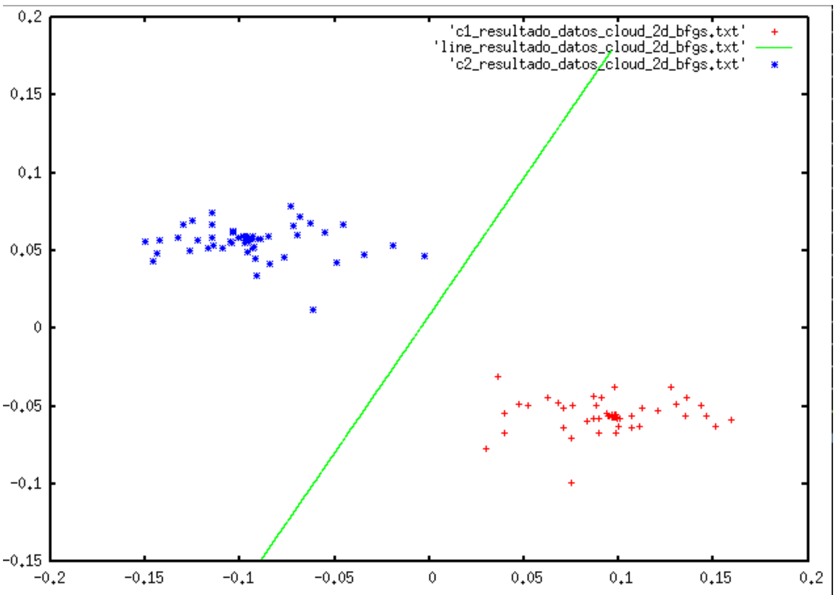

Figura 4: Nubes en 2d

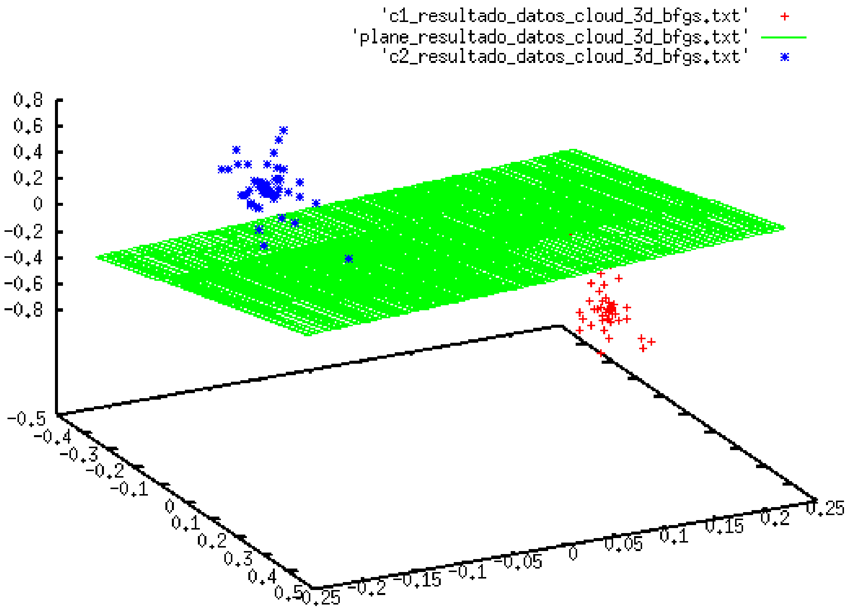

Figura 5: Nubes en 3d

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 incluído en la siguiente sección:

Funciones de vectores y matrices

```cpp
#include <vector>
#include <cmath>
#define sq(X) (X)*(X)
#define cub(X) (X)*(X)*(X)
typedef std::vector<std::vector<double> > matrix;
typedef std::vector<double> vector;

matrix cross(vector A, vector B)
{
  int n = A.size(), m = B.size();
  matrix C = std::vector<vector>(n, vector(m, 0));
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < m; j++)
      C[i][j] = A[i]*B[j];
  }
  return C;
}
matrix operator + (matrix A, matrix B)
{
  int n = A.size(), m = A[0].size();
  matrix C(n, vector(m));
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < m; j++)
      C[i][j] = A[i][j] + B[i][j];
  }
  return C;
}
vector operator * (matrix A, vector B)
{
  int n = A.size(), m = B.size();
  vector C(n, 0);

```

```
  for (int i = 0; i < n; i++)
  {
    for (int j = 0; j < m; j++)
      C[i] += A[i][j]*B[j];
  }

  return C;
}
vector operator + (vector A, vector B)
{
  int n = A.size();
  vector C(n);
  for (int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
  return C;
}
vector operator * (double eta, vector A)
{
  int n = A.size();
  vector ans(n);
  for (int i = 0; i < n; i++)
    ans[i] = eta*A[i];
  return ans;
}
vector operator - (vector A, vector B)
{
  int n = A.size();
  vector C(n);
  for (int i = 0; i < n; i++)
    C[i] = A[i] - B[i];
  return C;
}
double operator * (vector A, vector B)
{
  int n = A.size();
  double ans = 0;
  for (int i = 0; i < n; i++)
    ans += A[i]*B[i];
  return ans;
}

```

```
matrix operator * (matrix A, matrix B)
{
  int n = A.size(), m = B.size(), t = B[0].size();
  matrix C(n, vector(t));
  for (int i = 0; i < n; i++){
    for (int j = 0; j < t; j++) {
      C[i][j] = 0;
      for (int k = 0; k < m; k++)
        C[i][j] += A[i][k]*B[k][j];
    }
  }
```

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

```cpp
#include "vector_functions.cpp"
#include <iostream>

/**
 * Class Fuction:
 * clase abstracta para representar las funciones que
 * seran pasadas como parametro a la busqueda lineal
 * @N:          dimension del vector de entrada
 * @total:      numero de llamadas a la funcion
 * @total_d:    numero de llamadas a la derivada de la funcion
 * @total_h:    numero de llamadas al hessiando de la funcion
 * @total_i:    numero de veces que se calcula la inversa de
     una matriz
 */
class Function {
public:
    int N;
    int total;
    int total_d;
    int total_h;
    int total_i;
    /**
     * val - valor de la funcion evualuada en un punto
     * Parametros:
     *  x - vector en el que se evalua la funcion
     * Descripcion:
     *  retorna el valor de la funcion,
     *  por defecto es la funcion constate
     *  (igual a 1)
     */
    virtual double val(vector x){
        return 1;
    }

    /**
     * d - derivada
     * Parametros:
     *  x - vector, punto a evaluar la derivada
     * Descripcion:
```

```
     *  retorna el valor de la derivada en el
     *  punto dado
     */
    virtual vector d(vector x) {
        return vector(0);
    }
    /**
     * h - hessiano
     * Parametros:
     *  x - vector, punto a evaluar el hessiano
     *  retorna el valor del hessiano en el
     *  punto dado
     */
    virtual matrix h(vector x) {
        return matrix(0);
    }
    /**
     * inverse - inversa
     * Parametros:
     *  A - matriz de entrada
     * Descripcion
     *  retorna la inversa de la
     *  matriz dada como parametro
     */
    virtual matrix inverse(matrix &A){
        return matrix(0);
    }
};

class ET : public Function{
public:
    int M;
    matrix X;
    vector V, PHI, y, D;
    double rho;

    ET(int N0, matrix X, vector D, double rho = 1) :
        X(X), D(D), rho(rho) {
        N = N0;
        M = X.size();
        y = vector(M);
```

```
        PHI = vector(M);
    }
    double phi(double x) {
        x *= rho;
        return (exp(x) - exp(-x))/(exp(x) + exp(-x));
    }
    double dphi(double x) {
        x *= rho;
        return rho*sq(2/(exp(x) + exp(-x)));
    }
    double ddphi(double x) {
        return -2*rho*phi(x)*dphi(x);
    }
    void precalculations(vector w) {
        V = X*w;
        for (int k = 0; k < M; k++)
            y[k] = phi(V[k]);
        for (int k = 0; k < M; k++)
            PHI[k] = (y[k] - D[k])*dphi(V[k]);
    }
    virtual double val(vector w) {
        total++;
        precalculations(w);
        double e = 0;
        for (int k = 0; k< M; k++)
            e += sq(y[k]-D[k]);
        return e/2;
    }
    virtual vector d(vector w) {
        total_d++;
        matrix Xt = t(X);
        return Xt*PHI;
    }
    virtual matrix h(vector w) {
        total_h++;
        matrix H(N, vector(N));
        for (int i = 0; i < N; i++) {
            for (int j = i; j < N; j++){
                H[i][j] = 0;
                for (int k = 0; k < M; k++)
                    H[i][j] += X[k][i]*X[k][j]*(sq(dphi(V[k]))
```

```
                        + ddphi(V[k])*(y[k]-D[k]));
                H[j][i] = H[i][j];
            }
        }
        return H;
    }
    double det_adj(int i, int j, matrix &M) {
        matrix A(2, vector(2));
        int i2 = 0, j2 = 0;
        for (int i1 = 0; i1 < N; i1++) {
            j2 = 0;
            if (i1 == i) continue;
            for (int j1 = 0; j1 < N; j1++){
                if (j1 == j) continue;
                A[i2][j2] = M[i1][j1];
                j2++;
            }
            i2++;
        }
        return A[0][0]*A[1][1]-A[1][0]*A[0][1];
    }
    virtual matrix inverse(matrix &M) {
        total_i++;
        matrix X = matrix(N, vector(N));
        double det =    M[0][0]*(M[1][1]*M[2][2] - M[1][2]*M[2][1])
                      - M[0][1]*(M[1][0]*M[2][2] - M[1][2]*M[2][0])
                      + M[0][2]*(M[1][0]*M[2][1] - M[1][1]*M[2][0]);
        matrix Mt = t(M);
        for (int i = 0; i < N; i++){
            for (int j = 0; j < N; j++){
                X[i][j] = det_adj(i, j, Mt)*(((i+j)&1) ? -1 : 1);
                X[i][j] /= det;
            }
        }
        return X;
    }
};
```

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

```
#include "Function.cpp"
#include <climits>
#include <cstdlib>
#include <ctime>

/**
 * Class LinearSearch:
 * clase abstracta para representar las diferentes
 * busquedas lineales
 * @eps:    precision utilizada como condicion de parada
 * @MAX_IT: numero maximo de iteraciones permitidas
 * @f:      funcion utilizada para optimizar
 * @x:      valor del punto en cada iteracion
 * @gx:     valor del gradiente en cada iteracion
 * @fx:     valor de la funcion en cada iteracion
 */
class LinearSearch {
public:
    double eps;
    int MAX_IT;
    Function *f;
    vector x;
    vector gx;
    double fx;

    /**
     * alpha - longitud del paso
     * Parametros:
     *  @d - vector de la direccion de busqueda
     * Descripcion:
     *  Longitud del paso en el algoritmo iterativo,
     *  su valor por defecto es 1
     */
    virtual double alpha(vector &d)
    {
        return 1;
    }

    /**
```

```
     * dir - direccion de busqueda
     * Descripcion:
     *  Retorna la direccion de descenso
     *  del algoritmo, por defecto es la
     *  direccion opuesta al gradiente
     */
    virtual vector dir()
    {
        return -1*gx;
    }

    /**
     * linear_search
     * Parametros:
     *  @x0 - vector inicial
     * Descripcion:
     *  Algorimto de busqueda lineal,
     *  converge a la solucion que es
     *  almacenada en la variable x
     */
    int linear_search(vector x0)
    {
        vector d;
        x = x0;
        int k = 0;
        while (true) {
            k++;
            fx = f -> val(x);
            gx = f -> d(x);
            if (abs(gx) < eps || k > MAX_IT)
                break;
            d = dir();
            x = x + alpha(d)*d;
        }
        return k-1;
    }
};

/**
 * Class Cauchy
 * Subclase hija de LinearSearch
```

```
 * @al0:    paso inicial en el algoritmo de backtraking
 * gamma:   parametro del algoritmo de bactracking
 * @ro:     parametro del algoritmo de backtraking
 * @sig:    parametro que puede ser randomizado para
 *          variaciones de Cauchy, por defecto es igual a 1
 */
class Cauchy : public LinearSearch
{
public:
    double al0, gamma, ro, sig;

    /**
     * Cauchy- Constructor de la clase
     * Descripcion:
     *  Establece los parametros para el metodo de
     *  cauchy
     */
    Cauchy(double al0, double gamma, double ro, double sig = 1)
        :
        al0(al0), gamma(gamma), ro(ro), sig(sig) { }

    /**
     * Sigma -
     * Descripcion:
     *  retorna el valor del parametro sigma
     */
    virtual double sigma()
    {
        return sig;
    }

    /**
     * alpha - longitud del paso
     * Descripcion:
     *  sobrescribe la funcion de la clase padre
     *  utiliza el algoritmo de backtraking para
     *  para calcular la longitud del paso
     */
    virtual double alpha(vector &d)
    {
        double C = gamma*(gx*d), a = al0;
```

```
        while(f -> val(x + a*d) > fx + a*C){
            a = ro*a*sigma();
        }
        return a;
    }
};

/**
 * Class Cauchy
 * Subclase hija de Cauchy
 */
class Cauchy2 : public Cauchy {
public:

    /**
     * Cauchy2- Constructor de la clase
     * Descripcion:
     *  Establece los parametros para el metodo de
     *  cauchy e inicializa la semilla para generar
     *  los numeros aleatorios
     */
    Cauchy2(double al0, double gamma, double ro = 0.5, double
       sig = 1) :
        Cauchy(al0, gamma, ro, sig) {
            srand(time(NULL));
    }

    /**
     * sigma
     * Descripcion:
     *  sobrescribe la funcion de la clase padre
     *  utiliza el algoritmo de backtraking para
     *  para calcular la longitud del paso
     */
    virtual double sigma()
    {
        return (double) std::rand()/INT_MAX;
    }
};


```

```
/**
 * Class Newton
 * Subclase hija de Linear Search
 */
class Newton : public LinearSearch
{
    /**
     * dir - direccion de busqueda
     * Descripcion:
     *  sobrescribe la funcion de la clase padre
     *  calcular la direccion de busqueda
     */
    virtual vector dir() {
        matrix H = f -> h(x);
        H = f -> inverse(H);
        return -1*(H*gx);
    }
};

class Bfgs : public Cauchy
{
public:
```

Implementación de los generadores

```cpp
#include <iostream>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) + a;
}

int main() {
  double A, B, C;
  srand(time(NULL));
  A = get_rand(-3, 3);
  B = get_rand(-3, 3);
  C = get_rand(-3, 3);
  int N;
  double distance, error;

  cin >> N >> distance >> error;
  cout << N << endl;
  int i = 0;
  while(i < N) {
    double X, Y;
    X = get_rand(-10, 10);
    Y = get_rand(-10, 10);

    if (abs(A*X + B*Y + C) < distance)
      continue;
    cout << X + get_rand(-error/2, error/2) << ' ' << Y + get_rand(-error/2, error/2) << ' ';
    if (A*X + B*Y + C > 0)
      cout << 1;
    else
      cout << -1;
    cout << endl;
    i++;
```

```
    }
}
```

```
#include <iostream>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) + a;
}

int main() {
  double A, B, C, D;
  srand(time(NULL));
  A = get_rand(-3, 3);
  B = get_rand(-3, 3);
  C = get_rand(-3, 3);
  D = get_rand(-3, 3);

  int N;
  double distance, error;
  cin >> N >> distance >> error;
  cout << N << endl;

  int i = 0;
  while (i < N) {
    double X, Y, Z;
    X = get_rand(-10, 10);
    Y = get_rand(-10, 10);
    Z = get_rand(-10, 10);

    if (abs(A*X + B*Y + C*Z + D) < distance)
      continue;

    cout << X + get_rand(-error/2, error/2) << ' ' << Y+get_rand(-error/2, error/2) << ' ' << Z+get_rand(-error/2, error/2) << ' ';
    if (A*X + B*Y + C*Z + D > 0)
      cout <<  1;
    else
```

```
      cout << -1;
    cout << endl;
    i++;
  }
}
```

```
#include <iostream>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) + a;
}

vector <double> rand_vector(double r) {
  double x = get_rand(-1, 1), y = get_rand(-1, 1);
  double norm = sqrt(x*x + y*y);
  return vector <double>({x*r/norm, y*r/norm});
}

int main() {

  srand(time(NULL));
  int N;
  double distance, var;
  cin >> N >> distance >> var;
  default_random_engine generator;
  normal_distribution<double> distribution(0, var);
  vector <double> c1, c2;
  c1 = rand_vector(1);
  c2 = rand_vector(distance);
  c2[0] += c1[0];
  c2[1] += c1[1];

  cout << N << endl;

  for (int i = 0; i < N; i++){
    double X1, Y1, X2, Y2, r1, r2;
    vector <double> d1, d2;
    r1 = distribution(generator);
    r2 = distribution(generator);
    while (r1 > 1.5*var)
      r1 = distribution(generator);
```

```
    while (r2 > 1.5*var)
      r2 = distribution(generator);

    d1 = rand_vector(r1);
    d2 = rand_vector(r2);

    X1 = c1[0] + d1[0];
    Y1 = c1[1] + d1[1];
    X2 = c2[0] + d2[0];
    Y2 = c2[1] + d2[1];

    cout << X1 << '␣' << Y1 << '␣' <<  1 << endl;
    cout << X2 << '␣' << Y2 << '␣' << -1 << endl;
  }
}
```

```
#include <iostream>
#include <climits>
#include <cstdlib>
#include <ctime>
#include <algorithm>
using namespace std;

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) + a;
}

vector <double> rand_vector(double r) {
  double x = get_rand(-1, 1), y = get_rand(-1, 1), z = get_rand
      (-1, 1);
  double norm = sqrt(x*x + y*y + z*z);
  return vector <double>({x*r/norm, y*r/norm, z*r/norm});
}

int main() {

  srand(time(NULL));
  int N;
  double distance, var;
  cin >> N >> distance >> var;
  default_random_engine generator;
  normal_distribution<double> distribution(0, var);
  vector <double> c1, c2;
  c1 = rand_vector(1);
  c2 = rand_vector(distance);

  for (int i = 0; i < 3; i++)
    c2[i] += c1[i];

  cout << N << endl;

  for (int i = 0; i < N; i++){
    double X1, Y1, Z1, X2, Y2, Z2, r1, r2;

    vector <double> d1, d2;
    r1 = distribution(generator);
```

```
    r2 = distribution(generator);
    while (r1 > 1.5*var)
      r1 = distribution(generator);
    while (r2 > 1.5*var)
      r2 = distribution(generator);

    d1 = rand_vector(r1);
    d2 = rand_vector(r2);

    X1 = c1[0] + d1[0];
    Y1 = c1[1] + d1[1];
    Z1 = c1[2] + d1[2];

    X2 = c2[0] + d2[0];
    Y2 = c2[1] + d2[1];
    Z2 = c2[2] + d2[2];

    cout << X1 << '␣' << Y1 << '␣' << Z1 << '␣' <<  1 << endl;
    cout << X2 << '␣' << Y2 << '␣' << Z2 << '␣' << -1 << endl;
  }
}
```

Implementació de las funciones main que llaman a los algoritmos

```
#include "LinearSearch.cpp"
#include <iostream>
#include <fstream>
#include <ctime>
#include <climits>
#include <cstdlib>
#include <algorithm>
#define EPS 1e-3

void make_data(matrix &X, vector &D){
  vector x(3, 1);
  int M;
  std::cin >> M;
  D = vector(M);
  for (int i = 0; i < M; i++) {
    std::cin >> x[1] >> x[2] >> D[i];
    X.push_back(x);
  }
  double miuX = 0, varX = 0, miuY = 0, varY = 0;
  for (int i = 0; i < M; i++) {
    miuX += X[i][1];
    miuY += X[i][2];
  }
  miuX /= M;
  miuY /= M;
  for (int i = 0; i < M; i++) {
    varX += sq(X[i][1] - miuX);
    varY += sq(X[i][2] - miuY);
  }
  varX /= (M-1);
  varY /= (M-1);
  for (int i = 0; i < M; i++) {
    X[i][1] = (X[i][1] - miuX)/varX;
    X[i][2] = (X[i][2] - miuY)/varY;
  }
}
void make_graphics(vector w, matrix X, vector D){
  double C = w[0], B = w[2], A = w[1];
  std::ofstream line("line.txt"), c1("c1.txt"), c2("c2.txt");
```

```
  int N = X.size();
  double minY = 1e100, maxY = 1e-100, minX = 1e100, maxX = 1e-100;

  for (int i = 0; i < N; i++){
    minX = std::min(minX, X[i][1]);
    maxX = std::max(maxX, X[i][1]);
    minY = std::min(minY, X[i][2]);
    maxY = std::max(maxY, X[i][2]);
  }
  minX -= 0.05;
  maxY += 0.05;
  minY -= 0.05;
  maxY += 0.05;
  if (std::abs(A) > EPS){
    B /= A;
    C /= A;
    A = 1;
  }
  if (std::abs(B) < EPS) {
    line << -C/A << ' ' << minY << std::endl;
    line << -C/A << ' ' << maxY << std::endl;
  }else {
    int total = 10000;
    for (int i = 0; i < total; i++){
      double x0 = i*(maxX - minX)/total + minX, y0 = -(C + x0)/B;
      if (minY <= y0 && y0 <= maxY)
        line << x0 << ' ' << y0 << std::endl;
    }
  }
  for (int i = 0; i < N; i++) {
    if(D[i] == 1)
      c1 << X[i][1] << ' ' << X[i][2] << std::endl;
    else
      c2 << X[i][1] << ' ' << X[i][2] << std::endl;
  }
}
std::vector <int> results(vector w, matrix X, vector D) {
  int correct = 0, wrong = 0;
  int M = X.size();
```

```
  for (int i = 0; i < M; i++){
    double cross = w*X[i];
    if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
      correct++;
    else
      wrong++;
  }
  return std::vector<int>({correct, wrong});
}

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) - a;
}

int main() {
  matrix X;
  vector D;
  make_data(X, D);
  //ET *f = new ET(3, X, D, 0.5);
  vector  w0(3, 0);
  srand(time(NULL));
  int k;
  clock_t t1, t2;
  double ro = 0.3;
  Cauchy *cauchy = new Cauchy(10, 1e-4, ro);
  cauchy -> eps = EPS;
  cauchy -> MAX_IT = 1000000;
  std::cout << "Valor_de_rho␣rho_f_objetivo␣#llamadas_f_objetivo␣#llamadas_gradiente␣Norma_del_gradiente␣";
  std::cout << "#iteraciones␣Datos_bien_clasificados␣Datos_mal_clasificados␣Precisi n␣Tiempo_de_computo\n";
  for (double i=0.1; i<=1; i=i+0.1){
    ET *f = new ET(3, X, D, i);
    cauchy -> f = f;
    t1 = clock();
    k = cauchy -> linear_search(w0);
    t2 = clock();
    vector w = cauchy -> x;
    std::vector<int> r = results(w, X, D);
```

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

```
#include "LinearSearch.cpp"
#include <iostream>
#include <fstream>
#include <ctime>
#include <climits>
#include <cstdlib>
#include <algorithm>
#define EPS 1e-3

int N;
void make_data(matrix &X, vector &D){
  vector x(4, 1);
  int M;
  std::cin >> M;
  D = vector(M);
  for (int i = 0; i < M; i++) {
    std::cin >> x[1] >> x[2] >> x[3];
    std::cin >> D[i];
    X.push_back(x);
  }
  double miuX = 0, varX = 0;
  double miuY = 0, varY = 0;
  double miuZ = 0, varZ = 0;
  for (int i = 0; i < M; i++) {
    miuX += X[i][1];
    miuY += X[i][2];
    miuZ += X[i][3];
  }
  miuX /= M;
  miuY /= M;
  miuZ /=M;
  for (int i = 0; i < M; i++) {
    varX += sq(X[i][1] - miuX);
    varY += sq(X[i][2] - miuY);
    varZ += sq(X[i][3] - miuZ);
  }
  varX /= (M-1);
  varY /= (M-1);
  varZ /= (M-1);

  for (int i = 0; i < M; i++) {
```

```
    X[i][1] = (X[i][1] - miuX)/varX;
    X[i][2] = (X[i][2] - miuY)/varY;
    X[i][3] = (X[i][3] - miuZ)/varZ;
  }
}

void make_graphics(vector w, matrix X, vector D){
  double D1 = w[0], A = w[1], B = w[2], C = w[3];
  std::ofstream plane("plane.txt"), c1("c1.txt"), c2("c2.txt");
  int N = X.size();

  double minX = 1e100, maxX = -1e100, minY = 1e100, maxY = 1e
      -100, minZ = 1e100, maxZ = 1e-100;

  for (int i = 0; i < N; i++){
    minX = std::min(minX, X[i][1]);
    maxX = std::max(maxX, X[i][1]);
    minY = std::min(minY, X[i][2]);
    maxY = std::max(maxY, X[i][2]);
    minZ = std::min(minZ, X[i][3]);
    maxZ = std::max(maxZ, X[i][3]);
  }

  minX -= 0.1;
  minY -= 0.1;
  minZ -= 0.1;
  maxX += 0.1;
  maxY += 0.1;
  maxZ += 0.1;

  int total = 50;
  for (int i = 1; i <= total; i++) {
    for (int j = 1; j <= total; j++) {
      double x0 = i*(maxX - minX)/total + minX;
      double y0 = j*(maxY - minY)/total + minY;
      double z0 = -(D1 + A*x0 + B*y0)/C;
      if (minZ <= z0 && z0 <=maxZ)
        plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
    }
  }
  for (int j = 1; j <= total; j++) {
```

```
    for (int i = 1; i <= total; i++) {
      double x0 = i*(maxX - minX)/total + minX;
      double y0 = j*(maxY - minY)/total + minY;
      double z0 = -(D1 + A*x0 + B*y0)/C;
      if (minZ <= z0 && z0 <=maxZ)
        plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
    }
  }
  for (int i = 0; i < N; i++) {
    if(D[i] == 1)
      c1 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std
          ::endl;
    else
      c2 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std
          ::endl;
  }
}

std::vector <int> results(vector w, matrix X, vector D) {
  int correct = 0, wrong = 0;
  int M = X.size();
  for (int i = 0; i < M; i++){
    double cross = w*X[i];
    if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
      correct++;
    else
      wrong++;
  }
  return std::vector<int>({correct, wrong});
}
double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) - a;
}

int main() {
  matrix X;
  vector D;
  make_data(X, D);
  ET *f = new ET(4, X, D, 0.5);
  vector  w0(4, 0);
```

```
  srand(time(NULL));
  int k;
  clock_t t1, t2;
  double ro = 0.5;
  Cauchy *cauchy = new Cauchy(10, 1e-4, ro);
  cauchy -> eps = EPS;
  cauchy -> MAX_IT = 100000;
  cauchy -> f = f;
  t1 = clock();
  k = cauchy -> linear_search(w0);
  t2 = clock();
  vector w = cauchy -> x;
  std::vector<int> r = results(w, X, D);
  make_graphics(w, X, D);
  std::cout << "Valor de rho: " << ro << std::endl;
  std::cout << "Total de llamadas a: "<< std::endl;
  std::cout << "\tFuncion:   " << f -> total << std::endl;
  std::cout << "\tDerivadas: " << f -> total_d << std::endl;
  std::cout << "\tHessiano:  " << f -> total_h << std::endl;
  std::cout << "\tInversa:   " << f -> total_i << std::endl;
  std::cout << "Norma del gradiente:  " << abs(cauchy -> gx) << std::endl;
  std::cout << "Total de iteraciones: " << k << std::endl;
  std::cout << "Resultados: " << std::endl;
  std::cout << "\tDatos bien clasificados: " << r[0] << std::endl;
  std::cout << "\tDatos mal  clasificados: " << r[1] << std::endl;
  std::cout << "\tPrecision:               " << 100.00*r[0]/(r[0]+r[1]) << "%" << std::endl;
  std::cout << "Tiempo utilizado: " << (double) (t2 - t1)/CLOCKS_PER_SEC << std::endl;
}
```

```
#include "LinearSearch.cpp"
#include <iostream>
#include <fstream>
#include <ctime>
#include <climits>
#include <cstdlib>
#include <algorithm>
#define EPS 1e-3

void make_data(matrix &X, vector &D){
  vector x(3, 1);
  int M;
  std::cin >> M;
  D = vector(M);
  for (int i = 0; i < M; i++) {
    std::cin >> x[1] >> x[2] >> D[i];
    X.push_back(x);
  }
  double miuX = 0, varX = 0, miuY = 0, varY = 0;
  for (int i = 0; i < M; i++) {
    miuX += X[i][1];
    miuY += X[i][2];
  }
  miuX /= M;
  miuY /= M;
  for (int i = 0; i < M; i++) {
    varX += sq(X[i][1] - miuX);
    varY += sq(X[i][2] - miuY);
  }
  varX /= (M-1);
  varY /= (M-1);
  for (int i = 0; i < M; i++) {
    X[i][1] = (X[i][1] - miuX)/varX;
    X[i][2] = (X[i][2] - miuY)/varY;
  }
}
void make_graphics(vector w, matrix X, vector D){
  double C = w[0], B = w[2], A = w[1];
  std::ofstream line("line.txt"), c1("c1.txt"), c2("c2.txt");
  int N = X.size();
  double minY = 1e100, maxY = 1e-100, minX = 1e100, maxX = 1e
```

```
      -100;

  for (int i = 0; i < N; i++){
    minX = std::min(minX, X[i][1]);
    maxX = std::max(maxX, X[i][1]);
    minY = std::min(minY, X[i][2]);
    maxY = std::max(maxY, X[i][2]);
  }

  minX -= 0.05;
  maxY += 0.05;
  minY -= 0.05;
  maxY += 0.05;
  if (std::abs(A) > EPS){
    B /= A;
    C /= A;
    A = 1;
  }

  if (std::abs(B) < EPS) {
    line << -C/A << ' ' << minY << std::endl;
    line << -C/A << ' ' << maxY << std::endl;
  }else {
    int total = 10000;
    for (int i = 0; i < total; i++){
      double x0 = i*(maxX - minX)/total + minX, y0 = -(C + x0)/
          B;
      if (minY <= y0 && y0 <= maxY)
        line << x0 << ' ' << y0 << std::endl;
    }
  }
  for (int i = 0; i < N; i++) {
    if(D[i] == 1)
      c1 << X[i][1] << ' ' << X[i][2] << std::endl;
    else
      c2 << X[i][1] << ' ' << X[i][2] << std::endl;
  }
}
std::vector <int> results(vector w, matrix X, vector D) {
  int correct = 0, wrong = 0;
  int M = X.size();
```

```
  for (int i = 0; i < M; i++){
    double cross = w*X[i];
    if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
      correct++;
    else
      wrong++;
  }
  return std::vector<int>({correct, wrong});
}

double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) - a;
}

matrix random_matrix(int n){
  matrix id(n, vector(n));
  for (int i=0; i<n; i++){
    for (int j=0; j<n; j++){
      id[i][j]=get_rand(0, 5);
    }
  }
  id=id + t(id);
  id= id + n*identity(n);
  return id;
}

int main() {
  matrix X, I;
  vector D;
  make_data(X, D);

  ET *f = new ET(3, X, D, 0.5);
  vector  w0(3, 2);
  srand(time(NULL));
  for (int i = 0; i < w0.size(); i++)
    w0[i] = get_rand(-1, 1);
  int k;
  clock_t t1, t2;
  double ro = 0.3;

```

```
  I=identity(3);
  Bfgs *bfgs = new Bfgs(10, 1e-4, ro, I);
  bfgs -> eps=EPS;
  bfgs -> MAX_IT = 10000;
  bfgs -> f = f;
  t1 = clock();
  k = bfgs -> linear_search(w0);
  t2 = clock();
  vector w = bfgs -> x;
  std::vector<int> r = results(w, X, D);
  make_graphics(w, X, D);
  std::cout << "Valor de rho: " << ro << std::endl;
  std::cout << "Total de llamadas a: "<< std::endl;
  std::cout << "\tFuncion:   " << f -> total << std::endl;
  std::cout << "\tDerivadas: " << f -> total_d << std::endl;
  std::cout << "\tHessiano:  " << f -> total_h << std::endl;
  std::cout << "\tInversa:   " << f -> total_i << std::endl;
  std::cout << "Norma del gradiente:  " << std::abs(bfgs -> gx)
       << std::endl;
  std::cout << "Total de iteraciones: " << k << std::endl;
  std::cout << "Resultados: " << std::endl;
  std::cout << "\tDatos bien clasificados: " << r[0] << std::
      endl;
  std::cout << "\tDatos mal  clasificados: " << r[1] << std::
      endl;
  std::cout << "\tPrecision:               " << 100.00*r[0]/(r
      [0]+r[1]) << "%" << std::endl;
  std::cout << "Tiempo utilizado: " << (double) (t2 - t1)/
      CLOCKS_PER_SEC << std::endl;
}
```

```
#include "LinearSearch.cpp"
#include <iostream>
#include <fstream>
#include <ctime>
#include <climits>
#include <cstdlib>
#include <algorithm>
#define EPS 1e-3

int N;
void make_data(matrix &X, vector &D){
  vector x(4, 1);
  int M;
  std::cin >> M;
  D = vector(M);
  for (int i = 0; i < M; i++) {
    std::cin >> x[1] >> x[2] >> x[3];
    std::cin >> D[i];
    X.push_back(x);
  }
  double miuX = 0, varX = 0;
  double miuY = 0, varY = 0;
  double miuZ = 0, varZ = 0;
  for (int i = 0; i < M; i++) {
    miuX += X[i][1];
    miuY += X[i][2];
    miuZ += X[i][3];
  }
  miuX /= M;
  miuY /= M;
  miuZ /=M;
  for (int i = 0; i < M; i++) {
    varX += sq(X[i][1] - miuX);
    varY += sq(X[i][2] - miuY);
    varZ += sq(X[i][3] - miuZ);
  }
  varX /= (M-1);
  varY /= (M-1);
  varZ /= (M-1);
  for (int i = 0; i < M; i++) {
    X[i][1] = (X[i][1] - miuX)/varX;
```

```
    X[i][2] = (X[i][2] - miuY)/varY;
    X[i][3] = (X[i][3] - miuZ)/varZ;
  }
}
void make_graphics(vector w, matrix X, vector D){
  double D1 = w[0], A = w[1], B = w[2], C = w[3];
  std::ofstream plane("plane.txt"), c1("c1.txt"), c2("c2.txt");
  int N = X.size();
  double minX = 1e100, maxX = -1e100, minY = 1e100, maxY = 1e-100, minZ = 1e100, maxZ = 1e-100;
  for (int i = 0; i < N; i++){
    minX = std::min(minX, X[i][1]);
    maxX = std::max(maxX, X[i][1]);
    minY = std::min(minY, X[i][2]);
    maxY = std::max(maxY, X[i][2]);
    minZ = std::min(minZ, X[i][3]);
    maxZ = std::max(maxZ, X[i][3]);
  }
  minX -= 0.1;
  minY -= 0.1;
  minZ -= 0.1;
  maxX += 0.1;
  maxY += 0.1;
  maxZ += 0.1;

  int total = 50;
  for (int i = 1; i <= total; i++) {
    for (int j = 1; j <= total; j++) {
      double x0 = i*(maxX - minX)/total + minX;
      double y0 = j*(maxY - minY)/total + minY;
      double z0 = -(D1 + A*x0 + B*y0)/C;
      if (minZ <= z0 && z0 <=maxZ)
        plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
    }
  }

  for (int j = 1; j <= total; j++) {
    for (int i = 1; i <= total; i++) {
      double x0 = i*(maxX - minX)/total + minX;
      double y0 = j*(maxY - minY)/total + minY;
      double z0 = -(D1 + A*x0 + B*y0)/C;
```

```
      if (minZ <= z0 && z0 <=maxZ)
        plane << x0 << ' ' << y0 << ' ' << z0 << std::endl;
    }
  }

  for (int i = 0; i < N; i++) {
    if(D[i] == 1)
      c1 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std::endl;
    else
      c2 << X[i][1] << ' ' << X[i][2] << ' ' << X[i][3] << std::endl;
  }
}

std::vector <int> results(vector w, matrix X, vector D) {
  int correct = 0, wrong = 0;
  int M = X.size();
  for (int i = 0; i < M; i++){
    double cross = w*X[i];
    if ((cross > 0 && D[i] > 0) || (cross < 0 && D[i] < 0))
      correct++;
    else
      wrong++;
  }
  return std::vector<int>({correct, wrong});
}
double get_rand(double a, double b){
  double r = (double) std::rand()/INT_MAX;
  return r*(b - a) - a;
}

int main() {
  matrix X, I;
  vector D;
  make_data(X, D);
  ET *f = new ET(4, X, D, 0.5);
  vector  w0(4, 0);
  srand(time(NULL));
  int k;
  clock_t t1, t2;
```

```cpp
  double ro = 0.5;
  I=identity(4);
  Bfgs *bfgs = new Bfgs(10, 1e-4, ro, I);
  bfgs -> eps=EPS;
  bfgs -> MAX_IT = 10000;
  bfgs -> f = f;
  t1 = clock();
  k = bfgs -> linear_search(w0);
  t2 = clock();
  vector w = bfgs -> x;
  std::vector<int> r = results(w, X, D);
  make_graphics(w, X, D);
  std::cout << "Valor de rho: " << ro << std::endl;
  std::cout << "Total de llamadas a: "<< std::endl;
  std::cout << "\tFuncion:   " << f -> total << std::endl;
  std::cout << "\tDerivadas: " << f -> total_d << std::endl;
  std::cout << "\tHessiano:  " << f -> total_h << std::endl;
  std::cout << "\tInversa:   " << f -> total_i << std::endl;
  std::cout << "Norma del gradiente:  " << abs(bfgs -> gx) << std::endl;
  std::cout << "Total de iteraciones: " << k << std::endl;
  std::cout << "Resultados: " << std::endl;
  std::cout << "\tDatos bien clasificados: " << r[0] << std::endl;
  std::cout << "\tDatos mal  clasificados: " << r[1] << std::endl;
  std::cout << "\tPrecision:               " << 100.00*r[0]/(r[0]+r[1]) << "%" << std::endl;
  std::cout << "Tiempo utilizado: " << (double) (t2 - t1)/CLOCKS_PER_SEC << std::endl;
}
```

```makefile
CC = gcc
CXX = g++
C11 = -std=c++11
OPT = -Wall -O3 -Wno-unused-function -Wno-unused-variable

cauchy: src/cauchy.cpp
	$(CXX) $(C11) $< -o $@

cauchy3D: src/cauchy3D.cpp
```

```
	$(CXX) $(C11) $< -o $@

bfgs: src/bfgs.cpp
	$(CXX) $(C11) $< -o $@

bfgs3D: src/bfgs3D.cpp
	$(CXX) $(C11) $< -o $@

generator: src/generator.cpp
	$(CXX) $(C11) $< -o $@

generator3D: src/generator3D.cpp
	$(CXX) $(C11) $< -o $@

generator_clouds: src/generator_clouds.cpp
	$(CXX) $(C11) $< -o $@

generator_clouds3D: src/generator_clouds3D.cpp
	$(CXX) $(C11) $< -o $@


.PHONY: clean
clean:
	rm -fr cauchy cauchy3D bfgs bfgs3D generador generator
	    generator3D generator_clouds generator_clouds3D

cleanData:
	rm -fr *.txt
```