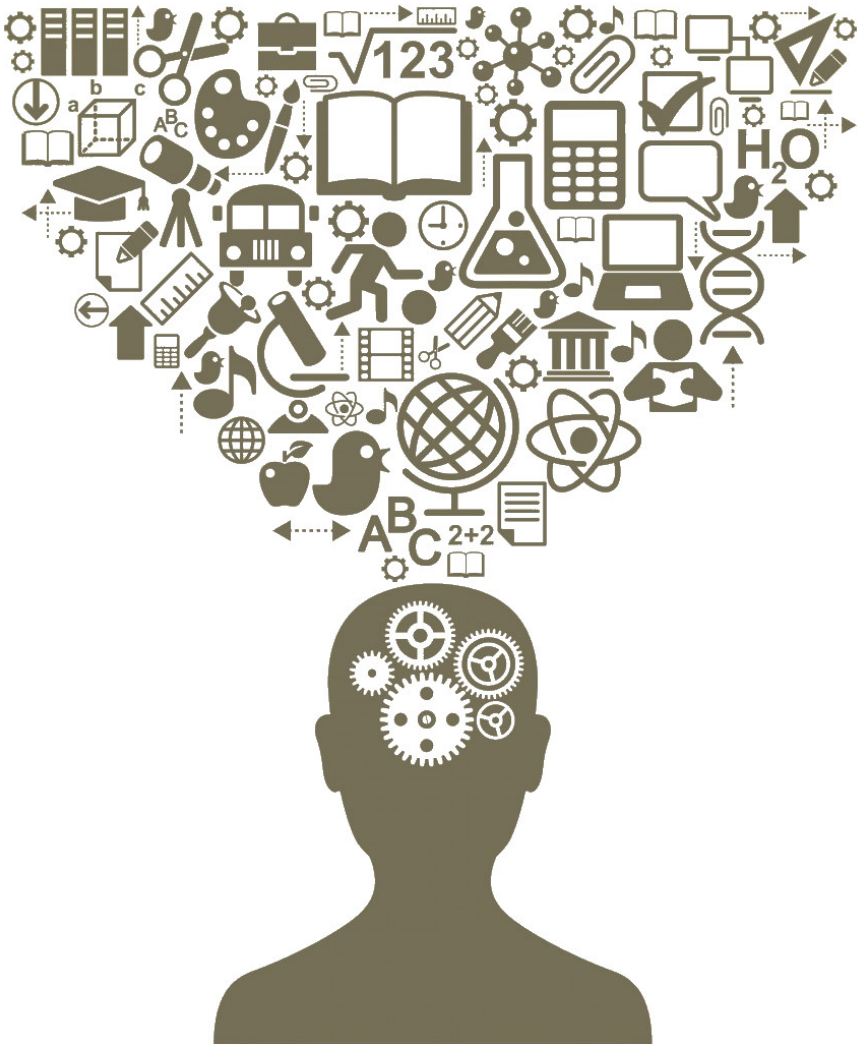
Razonamiento
Machine Learning

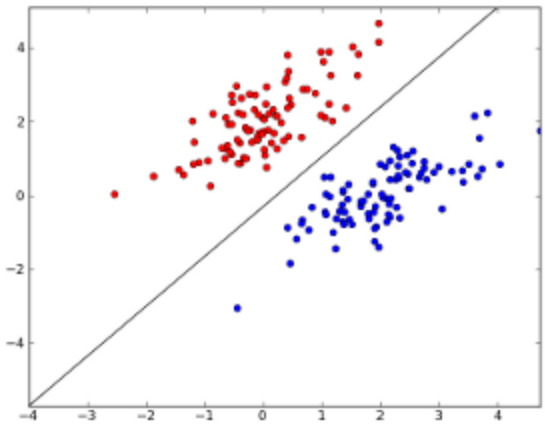


Salvador García Méndez
Javier Galván Martínez
Alberto Reales Diaz
Cesar Ivorra Oliver

Descripción de las tecnologías

Perceptrón

Propuesto por Rosenblatt en 1959, se trata de un modelo capaz de realizar tareas de clasificación de forma automática, a partir de un número de ejemplos etiquetados, el sistema determina la ecuación del hiperplano discriminante.

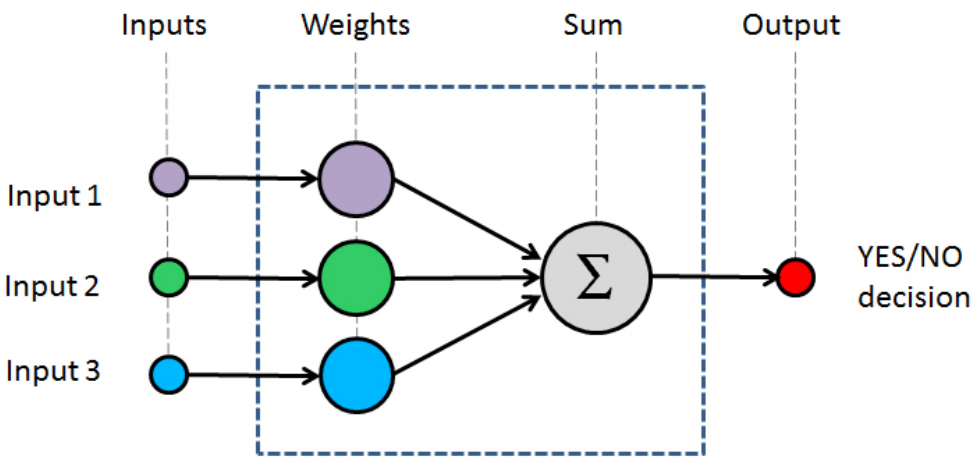


El perceptrón equivale a un hiperplano capaz de separar los prototipos en clases (de forma binaria en este caso). Si la salida del perceptrón es 0, la entrada pertenecerá a una clase y si es 1, la entrada pertenecerá a la clase contraria.

$$f(x) = \begin{cases} 1 & \text{si } w \cdot x - u > 0 \\ 0 & \text{en otro caso} \end{cases}$$

Donde *w* es un vector de pesos reales, *w·x* es el producto escalar y *u* es el umbral o bias, el cual representa el grado de inhibición de la neurona, es un término constante que no depende del valor que tome la entrada.

Un perceptrón presenta la siguiente disposición:



En esta estructura nos encontramos con las siguientes variables:

- Entrada *x(i)* denota el elemento en la posición *i* en el vector de la entrada
- El peso *w(i)* el elemento en la posición *i* en el vector de pesos
- El *y* denota la salida de la neurona
- El δ denota la salida esperada
- El α es una constante tal que $0 < \alpha < 1$

En este caso *alpha* se refiere a una tasa de aprendizaje que amortigua el cambio de los valores de los pesos. Y utilizaremos la siguiente regla de actualización de los pesos:

$$w(j)' = w(j) + \alpha(\delta - y)x(j)$$

En nuestro caso las variables de entrada y salida serán

```
int sClass;
double open;
double close;
double output;
```

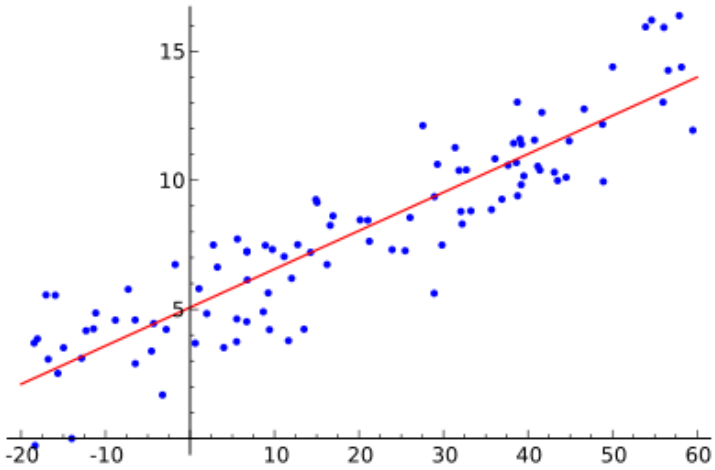
Mientras que no se cumpla un número máximo de iteraciones ajustaremos los pesos con las fórmulas vistas

```
while(iterations > 0)
{
    for (int i = 0; i < numData-1; ++i)
    {
            sClass = vClass[i+1];
            open = vOpen[i];
            close = vClose[i];
            output = 1;

            if (((vWeights[0] * open + vWeights[1] * close) - bias) < 0)
            {
                    output = -1;
            }

            if (output != sClass)
            {
                    vWeights[0] += alpha * (sClass - output) * open;
                    vWeights[1] += alpha * (sClass - output) * close;
                    bias +=  alpha * (sClass - output) * (-1);
            }
    }
    --iterations;
}
```

Regresión Lineal

Lo que tratamos con este modelo es, dado un conjunto de ejemplos determinar una función lineal que aproxime lo mejor posible a los valores deseados. La estimación de los parámetros supone encontrar la ordenada (en el origen o no) y la pendiente de una recta que mejor se aproxime a los puntos.

$$Y_t = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$



- *Yt* : variable dependiente, explicada o regresando.
- *X1, X2, …, Xp* : variables explicativas, independientes o regresores.
- *β1, β2, ..., βp*: parámetros, miden la influencia que las variables explicativas tienen sobre el regresando.
- *β0* es la intersección o término "constante".

Para determinar el modelo anterior, es necesario hallar (estimar) el valor de los coeficientes β1, β2, ..., βp. La linealidad en parámetros posibilita la interpretación correcta de los parámetros del modelo. Los parámetros miden la intensidad media de los efectos de las variables explicativas sobre la variable a explicar y se obtienen al tomar las derivadas parciales de la variable a explicar respecto a cada una de las variables explicativas:

$$\beta j = \delta Y / \delta Xj \; ; \; j = 1, \ldots, k$$

Nuestro objetivo es asignar valores numéricos a los parámetros β1, β2, ..., βk.

En caso de hacerlo solo con open y close como puntos 2D, podríamos aplicar la fórmula donde solo tendríamos *b0* y *b1*:

$$a = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$$
$$b = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

Esta fórmula tan simplificada apenas nos sirve para lograr una recta que interpola todos los puntos de la forma *X, Y*, lo cual, aunque en ocasiones de buenos resultados, tiene una capacidad expresiva altamente limitada.

```
for (int i = 0; i < numData-1; ++i)
{
    ySum += vY[i];
    xSum += vX[i];
    xySum += vX[i] * vY[i];
    xSumSqr += vX[i] * vX[i];
}

this->betaZero =
((ySum * xSumSqr) - (xSum * xySum)) / ((numData * xSumSqr) - (xSum * xSum));
this->beta1 =
((numData * xySum) - (xSum * ySum)) / ((numData * xSumSqr) - (xSum * xSum));
```

Para múltiples variables de entrada y así poder tener en cuenta tanto el open y close como la resta de los mismos, del modo *X1, X2, Y*. Encontramos para resolver los pesos, la forma escalar vista al inicio y la forma matricial, que es la que utilizaremos finalmente:

$$\begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \beta_0 + \beta_1 X_1 \\ \beta_0 + \beta_1 X_2 \\ \vdots \\ \beta_0 + \beta_1 X_n \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Así pues organizamos los datos de esa manera utilizando matrices de la librería *Armadillo.* Añadiendo también una columna inicial de constantes *1* ya que nos interesa para el valor de *beta0*.

```
mat X(numData-1, 3);

 for(int i = 0; i < numData-1; ++i){
     X(i,0) = 1;
     X(i,1) = vOpen[i];
     X(i,2) = vClose[i];
 }

 mat Y(numData-1, 1);

 for(int i = 0; i < numData-1; ++i)
     Y(i) = vClass[i+1];
```

A continuación aplicamos la fórmula que nos dará como resultado una matriz de pesos.

$$\mathbf{X'X} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ X_1 & X_2 & \cdots & X_n \end{bmatrix} \begin{bmatrix} 1 & X_1 \\ 1 & X_2 \\ \vdots & \vdots \\ 1 & X_n \end{bmatrix} = \begin{bmatrix} n & \sum X_i \\ \sum X_i & \sum X_i^2 \end{bmatrix}$$

$$(\mathbf{X'X})^{-1} = \frac{1}{n\sum X_i^2 - (\sum X_i)^2} \begin{bmatrix} \sum X_i^2 & -\sum X_i \\ -\sum X_i & n \end{bmatrix} = \frac{1}{nSS_X} \begin{bmatrix} \sum X_i^2 & -\sum X_i \\ -\sum X_i & n \end{bmatrix}$$

$$\mathbf{X'Y} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ X_1 & X_2 & \cdots & X_n \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} \sum Y_i \\ \sum X_i Y_i \end{bmatrix}$$

$$\mathbf{b} = (\mathbf{X'X})^{-1}\mathbf{X'Y} = \frac{1}{nSS_X} \begin{bmatrix} \sum X_i^2 & -\sum X_i \\ -\sum X_i & n \end{bmatrix} \begin{bmatrix} \sum Y_i \\ \sum X_i Y_i \end{bmatrix}$$

```
   //Obtenemos la traspuesta
    mat Xt = X.t();

    //Traspuesta por original
    mat XtX = Xt * X;

    // //Calculamos la inversa de X*Xt
    mat invXtX = XtX.i();

    // // Multiplicamos Xt * Y
    mat XtY = Xt * Y;

    // //Aplicamos la formula final ((Xt * X)^-1) * Xt*Y
    mat mBeta = invXtX * XtY;

    beta0 = mBeta(0,0);
    beta1 = mBeta(1,0);
    beta2 = mBeta(2,0);
```

En cuanto al cálculo del error que debemos minimizar, utilizaremos el descenso por gradiente. En primera instancia debemos obtener la función de coste asociada a este algoritmo.

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

```
double MultiLinearRegression::cost(int numData, std::vector<int> vClass, std::vector<double>
vOpen,std::vector<double> vClose){

    double cost = 0;
    double costAux = 0;
    for (int i = 0; i < numData-1; ++i)
    {
        costAux = calculate(vOpen[i],vClose[i]) - vClass[i+1];
        cost += costAux * costAux;
    }
    return cost/((double)(numData-1)*2);
}
```

```
double MultiLinearRegression::calculate(double x1, double x2){
    return (beta0 + beta1*x1 + beta2*x2);
}
```

Una vez obtenida iteramos de forma que los pesos se reajusten hasta minimizar el valor de J().

Repeat $\{$

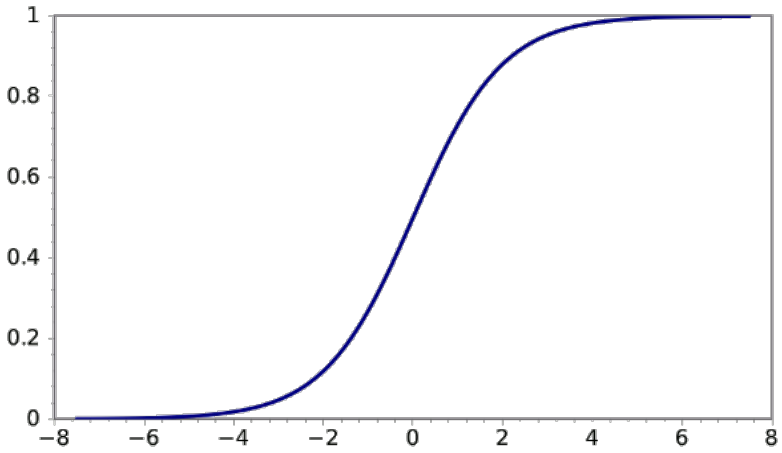
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ldots, \theta_n)$$

$\}$ (simultaneously update for every $j = 0, \ldots, n$)

```
for (int iters = 0; iters < iterations; ++iters){

    J = cost(numData, vClass, vOpen, vClose);

    if((abs(abs(oldJ) - abs(J))) < threshold){
           break;
    }

    fixingBeta0 = 0;
    fixingBeta1 = 0;
    fixingBeta2 = 0;
    costAux = 0;

    for(i = 0; i < numData-1; ++i){

           //h de theta
           costAux = calculate(vOpen[i],vClose[i]) - vClass[i+1];

           fixingBeta0 += costAux;
           fixingBeta1 += costAux * vOpen[i];
           fixingBeta2 += costAux * vClose[i];
    }

    fixingBeta0 -= (fixingBeta0/((double)(numData-1)))*eta;
    fixingBeta1 -= (fixingBeta1/((double)(numData-1)))*eta;
    fixingBeta2 -= (fixingBeta2/((double)(numData-1)))*eta;

    beta0 = fixingBeta0;
    beta1 = fixingBeta1;
    beta2 = fixingBeta2;

    oldJ = J;
}
```

Regresión Logística

Este modelo se basa en la obtención de la probabilidad de pertenencias de una clase según sus entradas de forma binomial, existen otras implementaciones que permiten diferenciar más de 2 clases, sin embargo ese modelo no lo aplicaremos en este problema. Su representación gráfica a diferencia de las regresion lineal y el perceptrón tiene forma de sigmoide dando lugar a valor entre 0..1 donde 0 y 1 representan la pertenencia a cada clase respectivamente. De esta forma al ascender o descender en el intervalo obtenemos la probabilidad de que sea una clase u otra.



El objetivo de nuestro modelo es predecir si la siguiente vez que se actualice la información de la bolsa nuestro algoritmo arroje un porcentaje de pertenencia a cada clase e indique al bot si debe comprar o debe vender.

La salida que obtenemos se basa en la siguiente función:

$$y = \frac{1}{1 + e^{-f(x)}}$$

De esta manera la función -f(x) sería el algoritmo real que deseamos alcanzar con el que se predice la bolsa, con lo que nosotros desarrollamos otra función que intente aproximar su salida a la de la función real.

Para ello nuestra hipótesis debe ajustar la línea que separa los dos conjuntos siguiendo la siguiente fórmula:

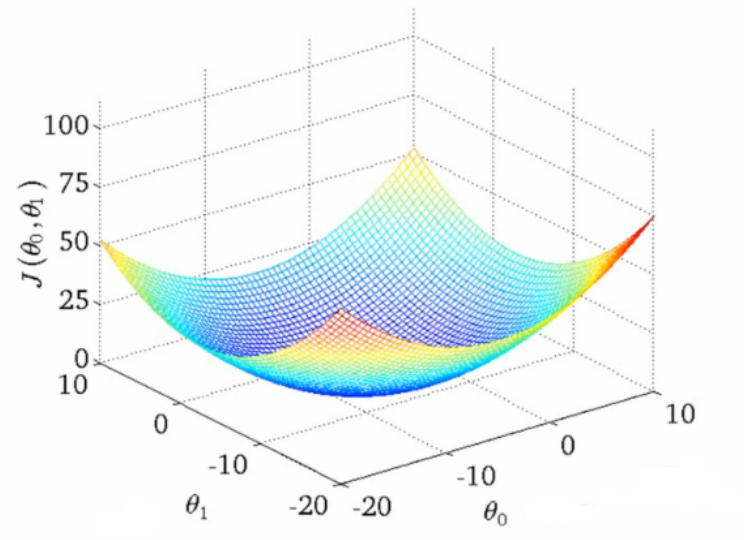
Training set: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

m examples $\quad x \in \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \quad x_0 = 1, y \in \{0, 1\}$

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

Donde X0 tendrá el valor de 1 y será independiente de los inputs y el valor de θ_0 será una constante que nosotros introduciremos de manera estática. El resto de inputs X será los que utilizamos como entradas y los valores de los pesos se inicializarán a 0.

Al igual que en la regresión lineal para ajustar la recta que nos separe el conjunto tendremos que calcular el coste de haber hecho un "salto" con el fin de encontrar el mínimo global, además nuestra función al ser convexa no tendrá mínimos locales sino que únicamente existirá un mínimo global. A continuación mostramos la forma que tiene la función.



Para calcular el coste aplicamos la siguiente fórmula:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

Donde esta fórmula la hemos implementado de la siguiente manera:

```
for (int i = 0; i < numData-1; ++i)
    {
        dClass = (vClass[i+1]+1)/2;
        probability = sigmoid(vOpen[i],vClose[i]);

        cost +=dClass * log(probability) + (1 - dClass) *  log(1 - probability);
    }
    cost *= (-1/ double (numData-1));
```

Con lo que obtenemos el coste del error cometido tras haber minimizado éstos con el reajuste de los pesos. Nuestro objetivo es hacer que el coste que se obtiene sea lo mas estable posible y como máximo difiera de un umbral indicado.

Para minimizar el error realizamos la siguiente operación:

1: Initialize the weights at $t = 0$ to $\mathbf{w}(0)$
2: **for** $t = 0, 1, 2, \ldots$ **do**
3: Compute the gradient

$$\nabla E_{\text{in}} = -\frac{1}{N} \sum_{n=1}^{N} \frac{y_n \mathbf{x}_n}{1 + e^{y_n \mathbf{w}^{\mathsf{T}}(t) \mathbf{x}_n}}$$

4: Update the weights: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E_{\text{in}}$
5: Iterate to the next step until it is time to stop
6: Return the final weights $\mathbf{w}$

Estas operaciones las hemos implementado de la siguientes manera:

```
peso0=peso1=pesosW=0;
pesosGood=vWeights;
pesoWGood= w0;
gradient=0;
for (int i = 0; i < numData-1; ++i)
{
     sClass = vClass[i+1];

    // Formula de Yaser para minimizar el Error
    gradient += (double)(sClass) / (1 + exp(sClass * w0));
    gradient += (double)(sClass * vOpen[i]) / (1 + exp(sClass * vWeights[0] * vOpen[i]));
    gradient += (double)(sClass * vClose[i]) / (1 + exp(sClass * vWeights[1] * vClose[i]));
}
gradient *= (double)-1/(double)(numData-1);
this->w0 -= eta*gradient;
vWeights[0] -= eta*gradient;
vWeights[1] -= eta*gradient;
```

Para concluir durante las iteraciones en las que vamos modificando los pesos y cálculos el coste asociado a esas modificaciones hacemos reajustes en el leerning rate, en el cual si hemos hecho un salto demasiado grande y nuestro coste ha aumentado significa que hemos saltado al otro lado de la pared y por tanto hemos hecho un mal salto, con lo que restablecemos los pesos a los pesos de la última iteración en la que se ha logrado un salto correcto y reducimos el learning rate para que haga un salto más pequeño.

Red Neuronal

Las redes neuronales combinan un determinado número de perceptrones (neuronas) distribuídos en diferentes capas para así obtener un clasificador fuerte a partir de los clasificadores débiles definidos por cada perceptron.

En nuestro caso particular, hemos creado una red con tres capas:

- 2 Neuronas en la primera capa
- 4 Neuronas en la capa oculta
- 1 Neurona en la capa de salida

Los pasos que seguimos en la ejecución son los siguientes:

1. Creamos una nueva red, cuyos perceptrones se inicializan con pesos aleatorios
2. Se hace el algoritmo de FeedFordward con los datos de entrada.
3. Aplicamos el algoritmo de BackPropagation para actualizar los pesos de las neuronas
4. Validamos el algoritmo

Esta serie de pasos se repite tantas veces como divisiones se hayan hecho del conjunto de datos en el cross-validation.

FeedFordward

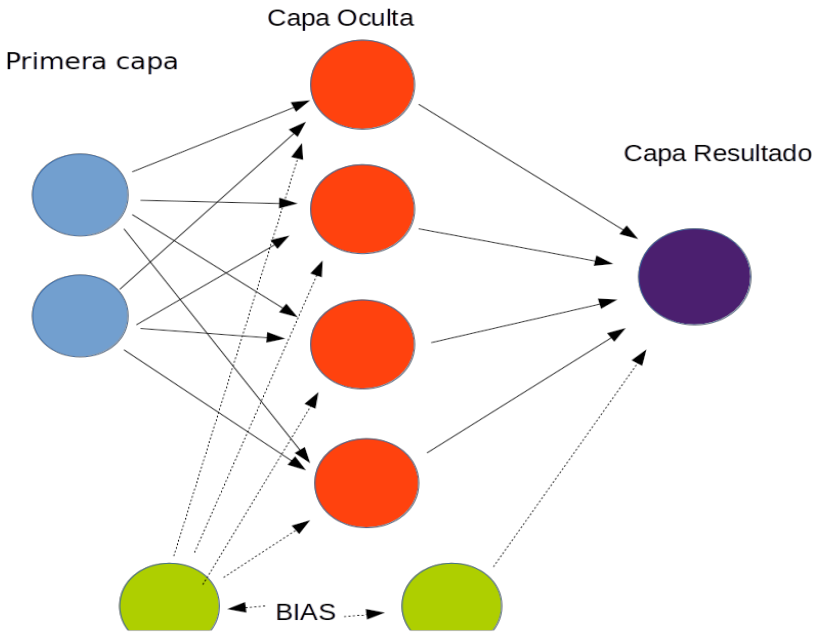
Este algoritmo hace que los valores de entrada recorran la red de la capa más exterior hasta la capa de salida, simplemente haciendo que la salida de un perceptrón componga la entrada de cada uno de los siguientes.

BackPropagation

Este algoritmo es el más utilizado para actualizar los valores de los pesos de cada neurona. Lo que se hace es recorrer la red neuronal en sentido inverso e ir actualizando los pesos de las neuronas en función de la salida que debería haber dado. Esto se hace con cada línea de datos para entrenar el modelo.

Topología

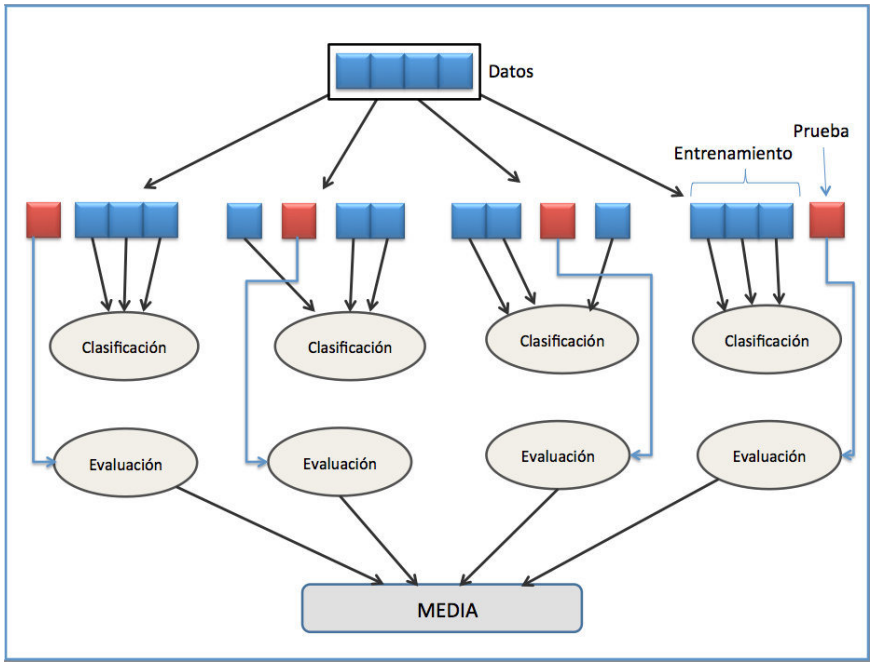
Además de las neuronas antes descritas, cada capa cuenta con una neurona extra, la llamada neurona "Bias". Esta neurona se conecta directamente con todas las neuronas de la siguiente capa y no recibe ninguna entrada. Lo que hace es añadir en peso extra a cada neurona con el objetivo de normalizar los resultados. De esta forma, "Bias" actúa como *Learning Rate* y regula la velocidad a la que la red se adapta a la función objetivo.



Como añadido, hemos hecho que la topología de la Red Neuronal sea adaptativa. Es decir, en el constructor, podemos añadirle tantas capas de neuronas como queramos. Esto lo hemos hecho para poder hacer experimentos y probar si con diferentes configuraciones obtenemos diferentes resultados. Desgraciadamente, no hemos tenido tiempo para hacer pruebas más exhaustivas.

ADDON01 - Cross Validation

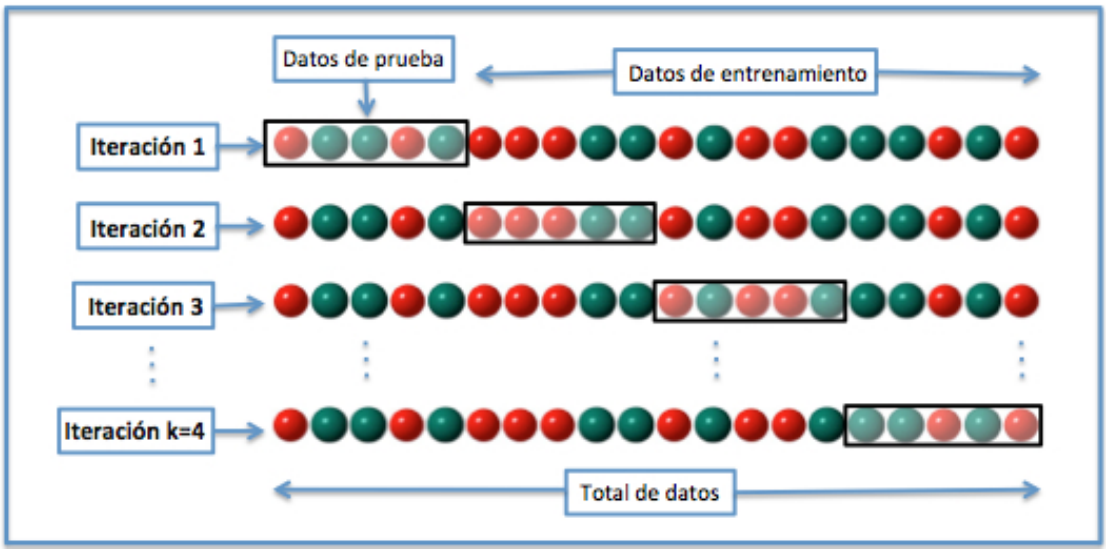
Cross-validation es una técnica utilizada para evaluar los resultados de un análisis estadístico y garantizar que son independientes de la partición entre datos de entrenamiento y prueba. Consiste en repetir y calcular la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones.



Existen distintos tipos de cross-validation, pero para nuestro caso hemos elegido el K-iteraciones.

En cross-validation de K iteraciones o *K-fold cross-validation* los datos de muestra se dividen en K subconjuntos. Uno de los subconjuntos se utiliza como datos de prueba y el resto (K-1) como datos de entrenamiento. El proceso de validación cruzada es repetido durante k iteraciones, con cada uno de los posibles subconjuntos de datos de prueba. Finalmente se realiza la media aritmética de los resultados de cada iteración para obtener un único resultado. Este método es muy preciso puesto que evaluamos a partir de K combinaciones de datos de entrenamiento y de prueba, pero aun así tiene una desventaja, y es que, a diferencia del método de retención, es lento desde el punto de vista computacional.

En la práctica, la elección del número de iteraciones depende de la medida del conjunto de datos.



Cross-Validation separa el conjunto de entrenamiento es un número determinado de folds y posteriormente va combinando los diferentes k-fold para tener diferentes pruebas de entrenamiento y test. Lo hace de la siguiente manera:

```
K-fold CV es
Divide los n ejemplos en k conjuntos disjuntos (folds)
Para i=1 hasta k
       Entrenar usando todos los folds menos i
       Emplear el fold i para estimar el error
Devolver la media de los errores de los k folds
Fin
```

Manual de utilización

Prerrequisitos

Para ejecutar correctamente el software desarrollado necesitaremos tener instaladas en nuestro sistema la librería *Armadillo* y sus dependencias *Lapack, Boost* y *Blas.* Amén de las herramientas propias para estos proyectos tales como *GNU C++*, etc. Podemos obtener las librerías mencionadas podemos hacerlo a través de la terminal escribiendo:

sudo apt-get install liblapack-dev
sudo apt-get install libblas-dev
sudo apt-get install libboost-dev
sudo apt-get install libarmadillo-dev

Ejecución

Una vez compilado, podemos ejecutar programa de forma directa con acceso a todas las opciones utilizando el siguiente comando en la terminal:

./cpcr2Machine [fichero.data] [numAlgorithm] [plot] [n-iters] [learning-rate]

En cuanto al parámetro numAlgorithm, se puede elegir entre los siguientes:

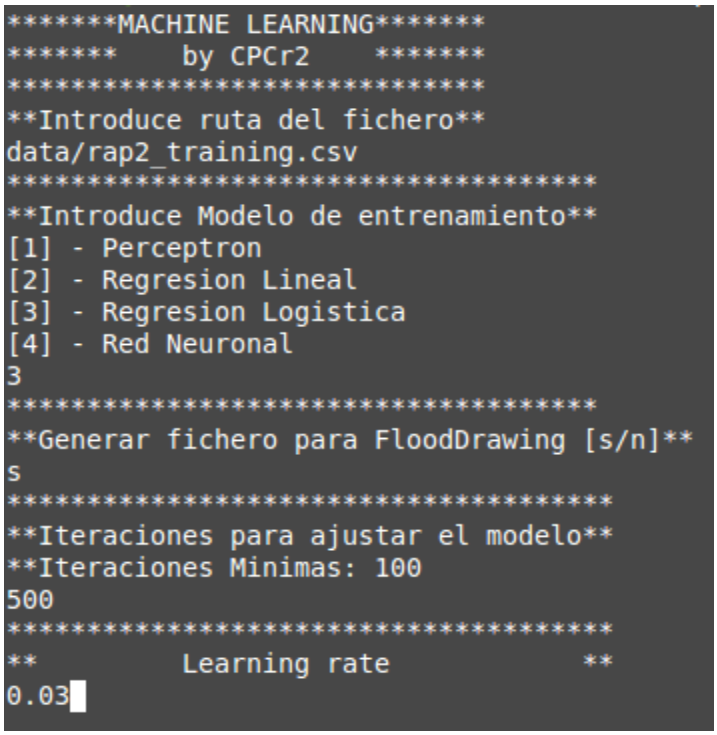
1. Perceptrón
2. Regresión Lineal
3. Regresión Logística
4. Red Neuronal

El parámetro plot puede ser 's' o 'n', si queremos que el sistema genere un archivo.data con todos los puntos en el rango del aprendizaje, clasificados para dibujarlos con *GNUPLOT* y obtener una aproximación de las gráficas, un archivo por cada ejecución del cross-validation.

Los dos últimos parámetros sirven para indicar el número de iteraciones máximo que tendrán los algoritmos de aprendizaje y el factor de aprendizaje, respectivamente.

Ejemplo: *./cpcr2Machine data/rap2_training.data 4 n 500 0.3*

De manera opcional, podemos optar por ejecutar el programa, simplemente, con el comando *./cpcr2Machine*, sin pasarle ningún argumento. Si hacemos esto, entonces el programa nos mostrará un menú, donde nos irá pidiendo que introduzcamos uno a uno los valores para los parámetros, como puede verse en la siguiente imagen:



Menú de ejecución

Uso del bot

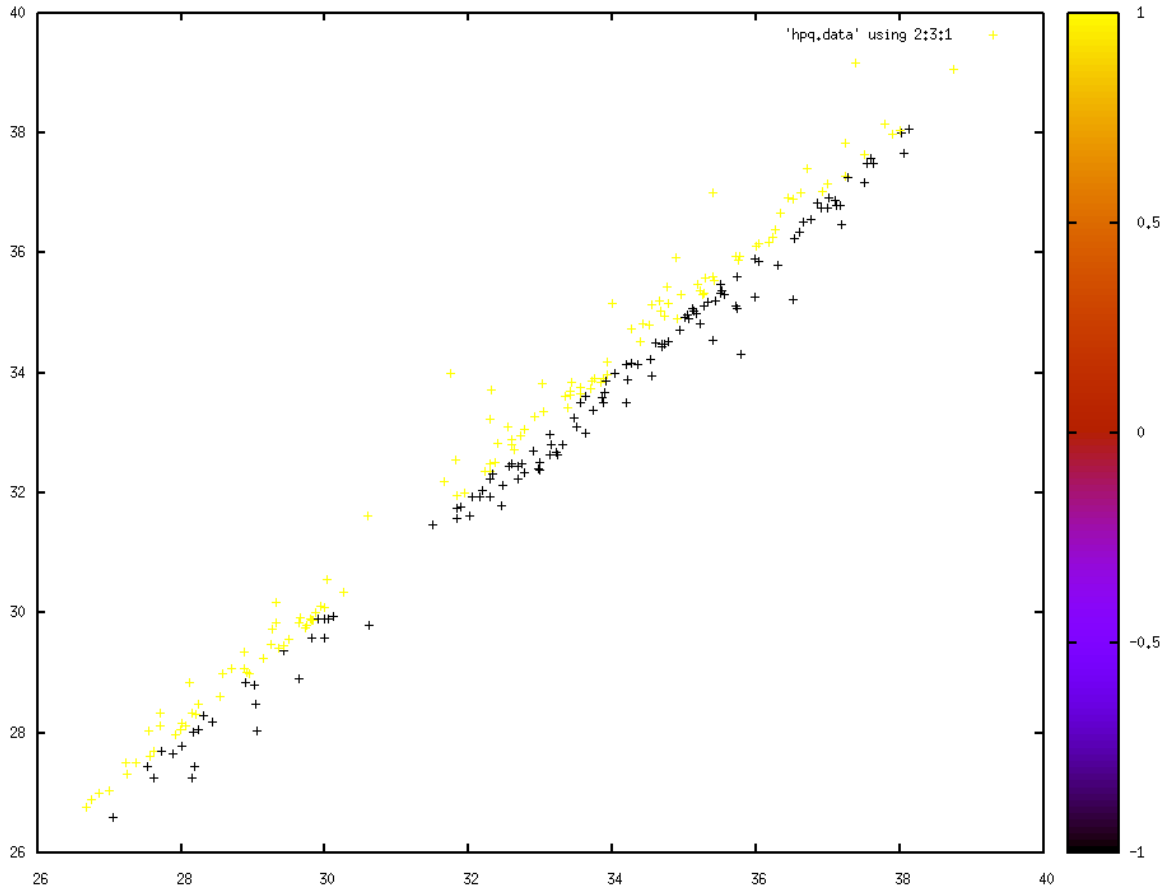
Para incluir nuestros algoritmos en el bot, lo único que habría que hacer es crear un objeto de la clase correspondiente al algoritmo que queremos utilizar, con los parámeteros apropiados, y después utilizar el método *.validate()* en el método *sendCommand()* del bot.

Ejemplo:

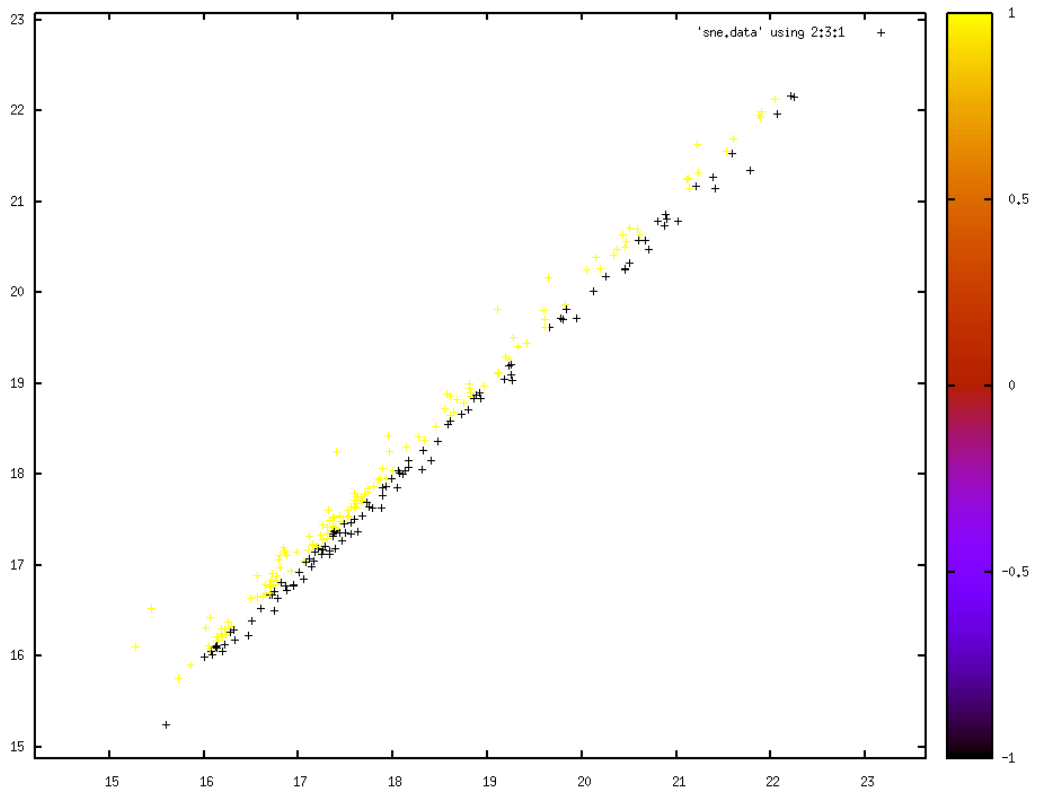
```
LogisticRegression lr = LogisticRegression(-1.03159, -0.0315945, -0.0315945);
double c = (double)(rand() % 1000) / 999.0;

int check = lr.validate(open, close);
```

Experimentación

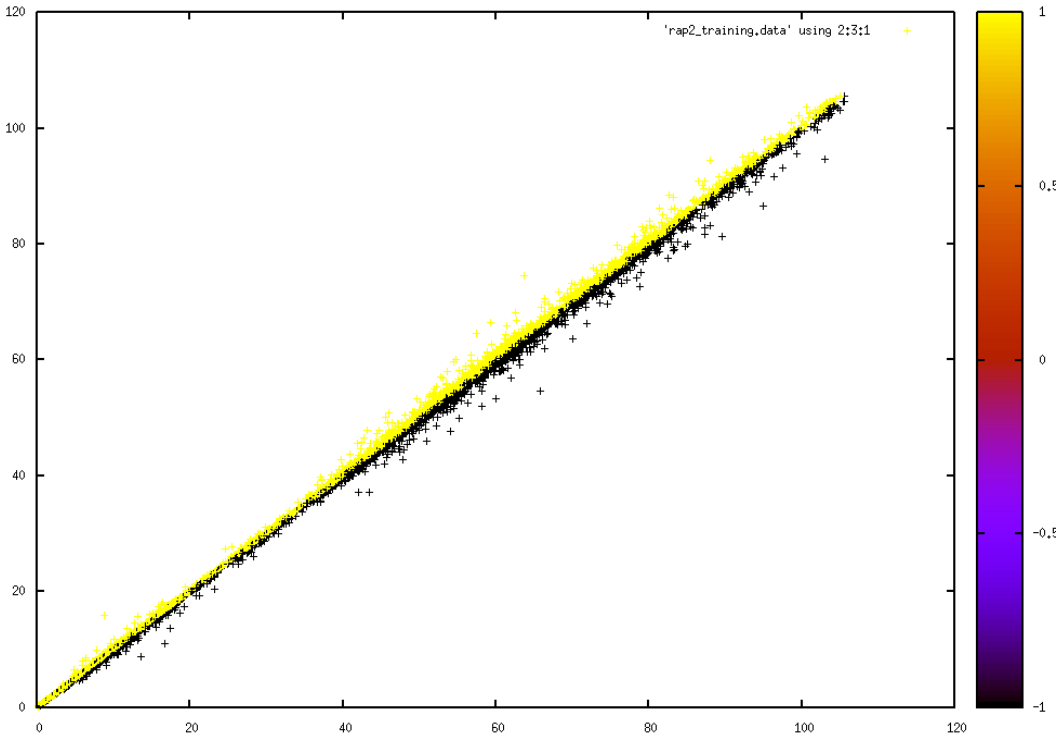
Inicialmente descargamos los datos de bolsa perteneciente a los 251 últimos días de la empresa *Hewlett-Packard y Sony.* Más adelante introdujimos *rap2_training.* Organizados de forma *"Sube/Baja, Apertura, Cierre",* como veremos a continuación el primer parámetro indica el color de la clase y en base a los dos siguientes se sitúan los puntos en la gráfica. Como veremos más adelante para tener una idea más o menos aproximada de la forma en la que separan los conjuntos de datos los algoritmos desarrollados, hemos incluido una función de pintado por inundación. Utilizando *GNUPLOT.*



Hewlett-Packard



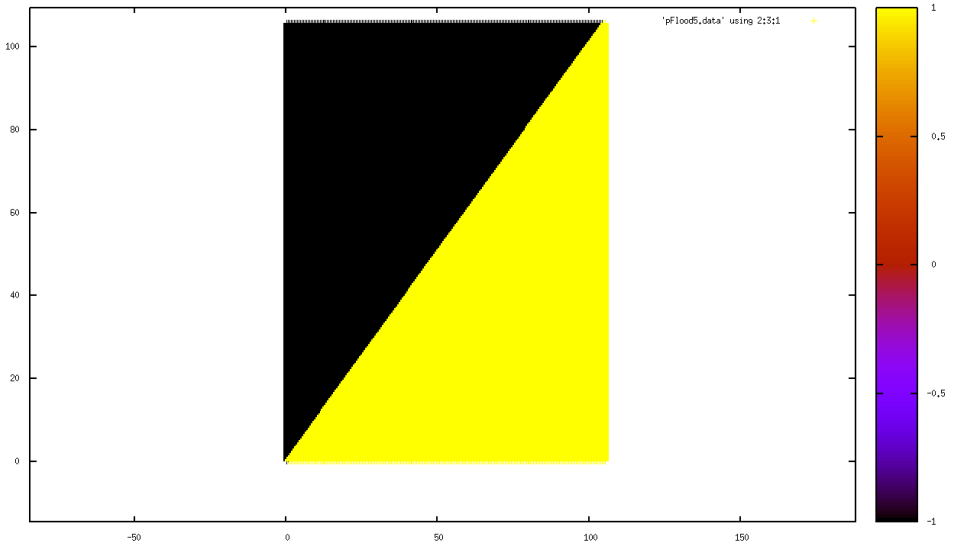
Sony



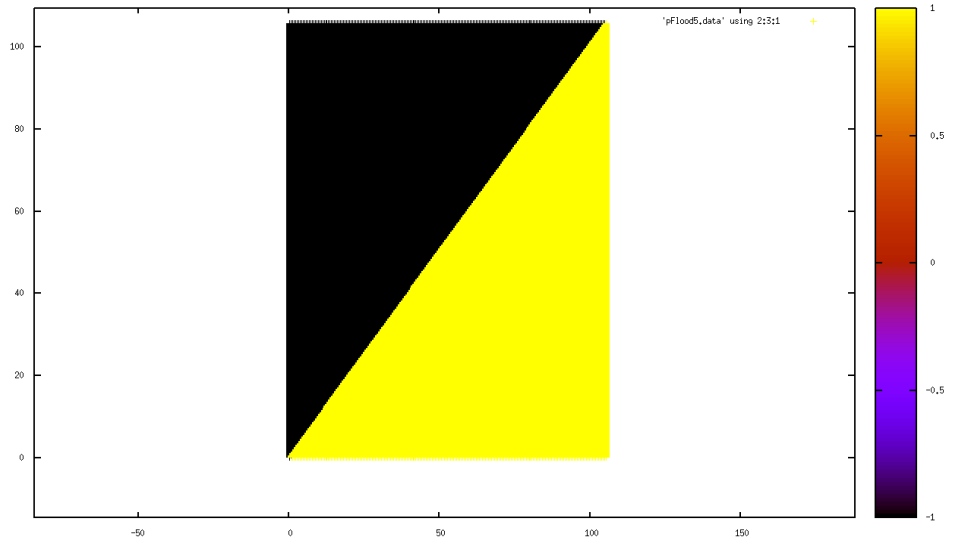
rap2_training

Perceptrón

Fran.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.03	1000	50,084%	49,91%
Pueba2	0.00000002	100000	49,845%	50,15%

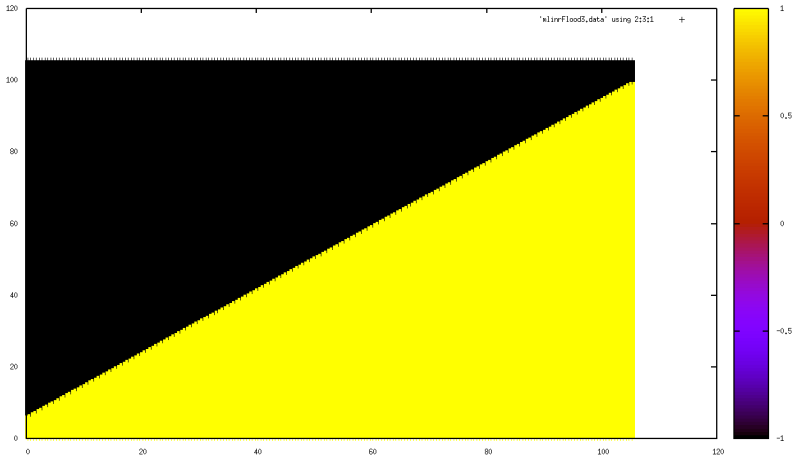


Prueba1

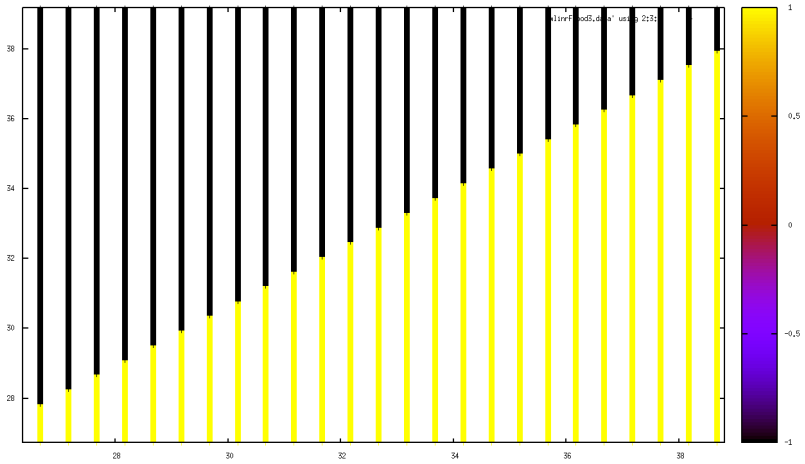


Prueba2

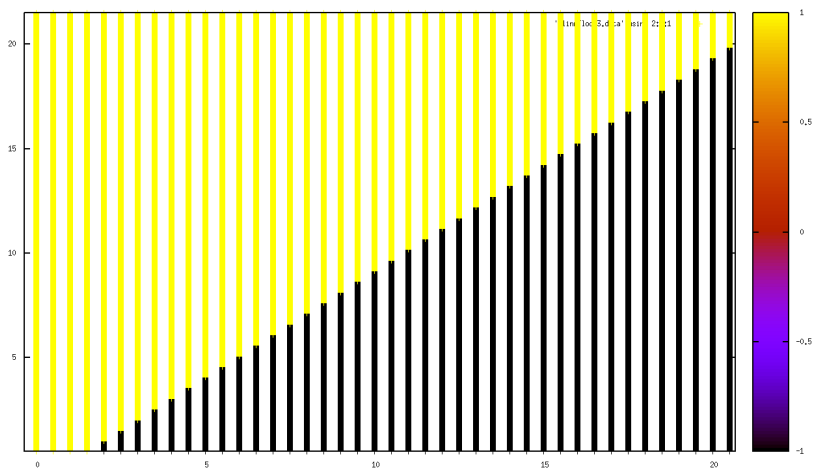
Regresión Lineal



Fran.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.003	100000	46.642%	53.368%



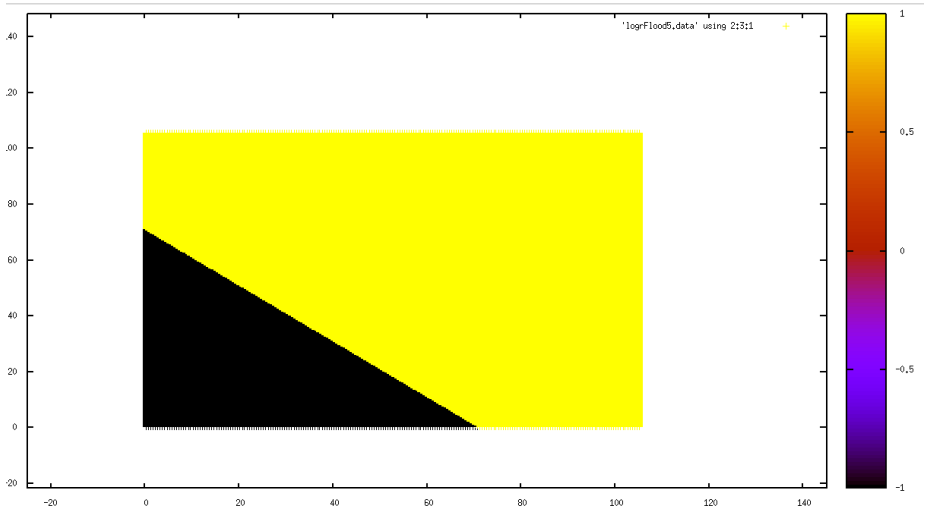
Hpq.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.003	100000	50.8%	51.2%



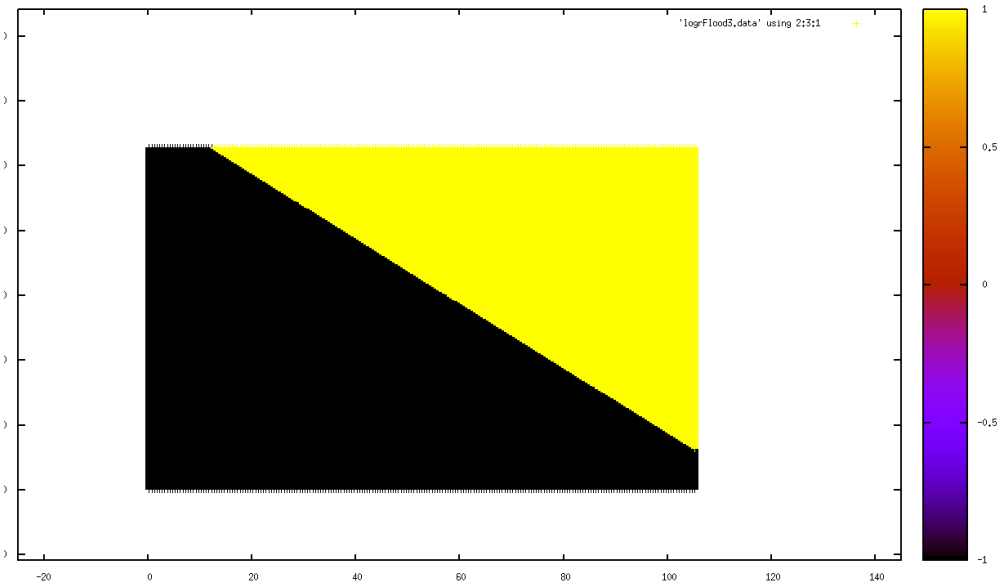
Sne.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.003	100000	46.2745%	55.6863%

Regresión Logística

Fran.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.0004	10000	46.744%	53.266%
Prueba2	0.003	1000	46.986%	53.024%

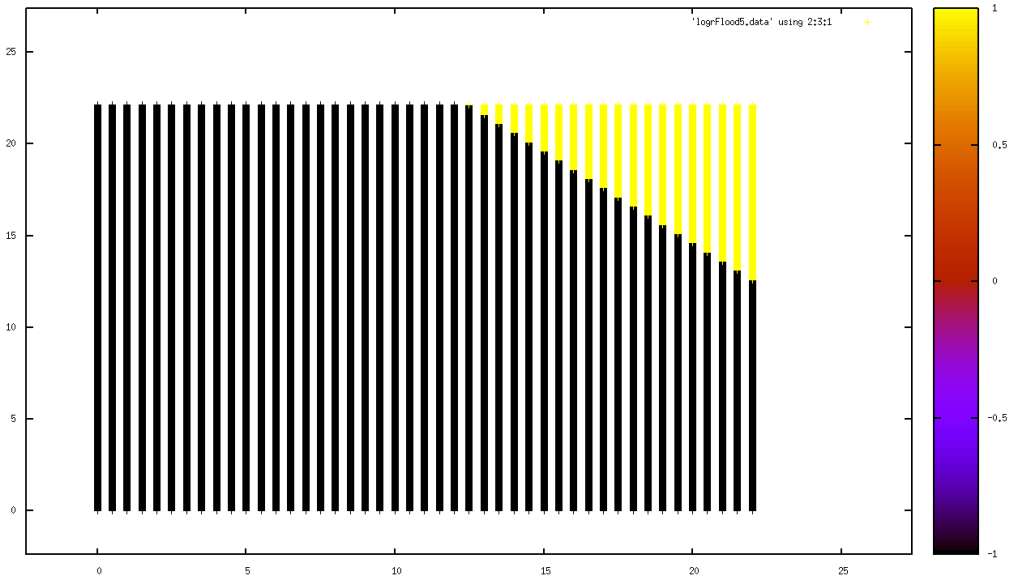


Prueba1

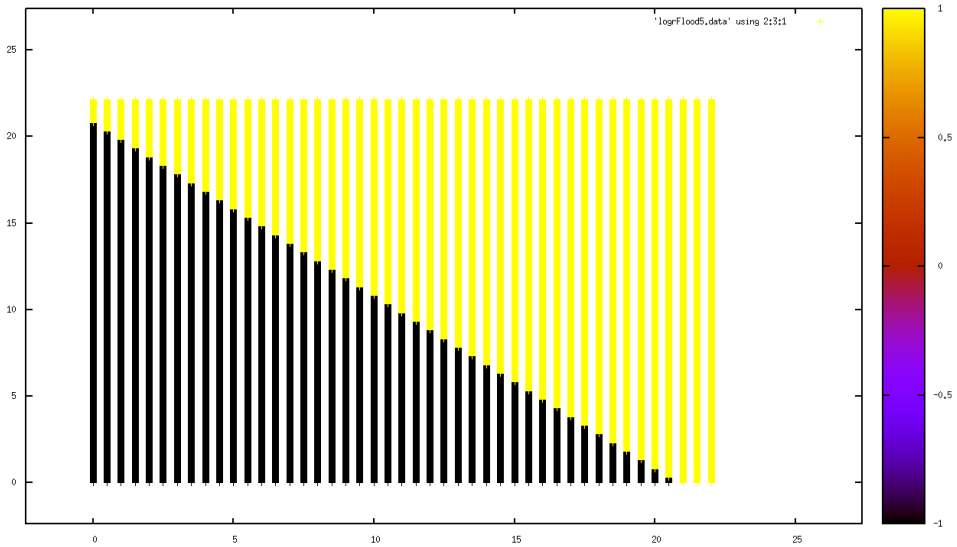


prueba2

Sne.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.0000000000007	10000	55,2941%	44.7059%
Prueba2	0.000000000001	10000	49.0196%	50,9804%



Prueba 1



Prueba2

Red Neuronal

sne.csv	Alpha	Iteraciones	Aciertos	Error
Prueba1	0.03	500	56.86 %	45.098 %
Prueba2	1	500	52.84 %	47.16 %
Prueba3	0.15	1000	50.05 %	49.95 %

Conclusiones

Tras el desarrollo de este práctica nuestras conclusiones son las siguientes:

- Curva de aprendizaje elevada. Desde un primer momento y durante el desarrollo de los modelos de machine learning hemos tenido que estudiar muy cuidadosamente todas las expresiones matemáticas que se nos planteaban en los videos proporcionados por el profesor los cuales no bastaba con ver una única vez.

- Dificultad con librerías externas. El uso de librerías como Armadillo han hecho que hayamos tenido problemas con las compilaciones y dependencias de éstas para poder tener un ejecutable en condiciones.

- Finalmente a nivel personal nos ha ayudado a comprender mejor cómo funcionan los distintos algoritmos existentes de machine learning, y las formas que hay para poder entrenarlos y ajustarlos a los datos con los que se pretende que nuestro modelo sea capaz de proporcionarnos una salida en base a todas las acciones ocurridas en el pasado.

Referencias

Linear Regression in Matrix Form
Stanford Machine Learning
Machine Learning Video Library
Installing Armadillo
Cplusplus