Aprendizaje Automático
(Machine Learning) por Andrew Ng

Stanford University - Coursera.org

I. Introducción

El aprendizaje automático es una rama de la ***Inteligencia Artificial*** (AI).

Aprendizaje Automático: *Es el campo de estudio que le da a las computadoras la habilidad de aprender sin ser programada explícitamente* (Arthur Samuel - 1959).

También puede definirse como un programa computacional que aprende desde la **experiencia (E)** con respecto a algunas **tareas (T)** y aumenta su **rendimiento (P)** a causa de la experiencia.

Ejemplo. Autodetección de Spam.
- E : Observar los correos marcados como spam y cuáles no lo fueron.
- T : Clasificación de los correos.
- P : Porcentaje de correos clasificados correctamente.

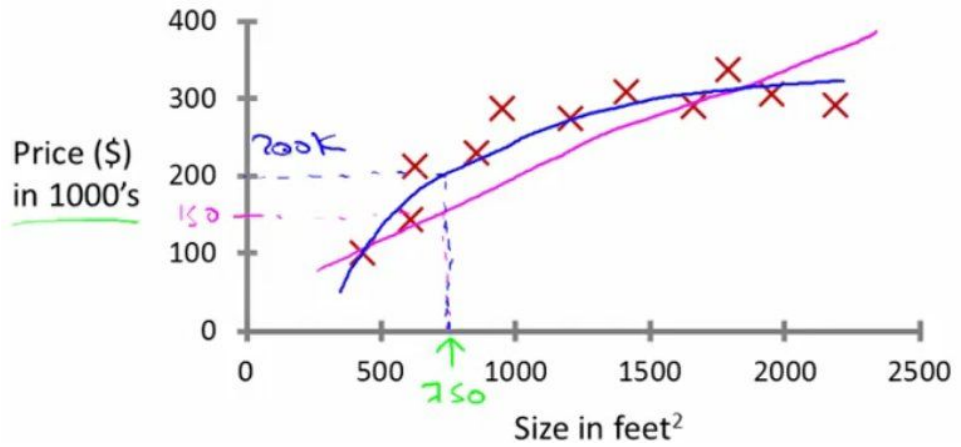
Tipos de aprendizaje

- Aprendizaje Supervisado (*Supervised Learning*)
- Aprendizaje No Supervisado (*Unsupervised Learning*)
- Aprendizaje Reforzado (*Reinforcement Learning*)
- Sistemas de recomendación (*Recommender Systems*)

Aprendizaje Supervisado

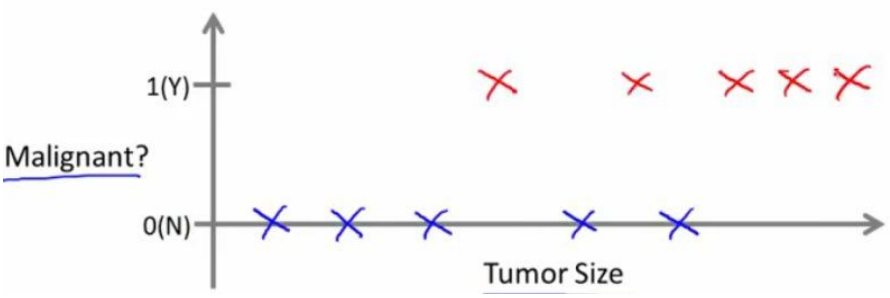
Ejemplo de REGRESIÓN

Se quiere conocer el precio de una casa de 750 m^2, pero se tiene información sobre casas de tamaños similares. Entonces lo que se hace es elegir la función correcta que muestra el comportamiento de los precios respecto al tamaño de la propiedad.



Este método parte de un conjunto de "respuestas correctas" y tratar, a partir de éstas, generar más "respuestas correctas". Esto se conoce como **regresión** (*predecir el resultado de un valor continuo - precio*).

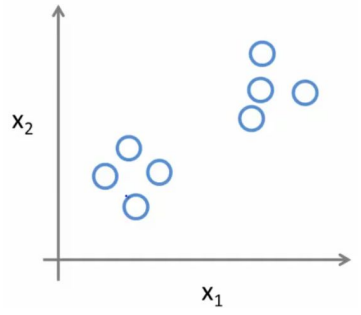
Ejemplo de CLASIFICACIÓN



Se desea analizar si un tumor sea maligno o benigno de acuerdo a su tamaño.

Se tomaron 5 muestras por cada posibilidad.
Esto se conoce como problema de **clasificación** (*se intenta predecir un valor 0 ó 1, o mayor*).
Para mejorar esta estimación podemos agregar más variables, pero ello requiere un consumo de recursos mayor. Para esto se utiliza una **Máquina de Soporte Vectorial (SVM)**.

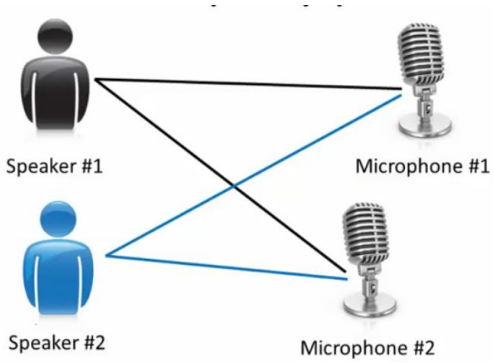
Aprendizaje No Supervisado



En este tipo de aprendizaje no hay diferencias entre los datos, sólo se trata de encontrar diferencias o agrupaciones.
A esto se lo conoce como **Algoritmo de Agrupamiento**.

Este algoritmo se utiliza en *Google News*.

Ejemplo. Problema de la fiesta de cóctel.



Se da una situación donde dos personas hablan al mismo tiempo.
Se colocan dos micrófonos a distintas distancias para grabar sus voces pero se desea prestar atención sólo a una.

El algoritmo trata de separar las dos fuentes de ruido.

Para implementarlo, se utiliza una sola línea de código:

```
[w,s,v] = svd( (repmat(sum(x.*x,1) , size(x,1) ,1)
      .*x) *x');
```

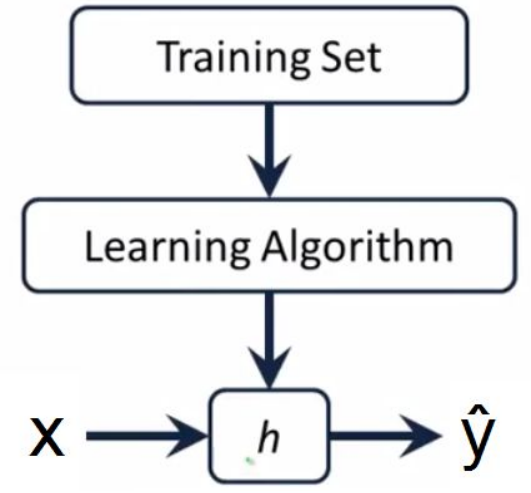
Se aplica **descomposición en valores singulares**.

II. Regresión lineal con una variable

La regresión lineal es el método que se aplica en el aprendizaje supervisado a las funciones continuas para encontrar la "*respuesta correcta*" a partir de un conjunto de "*soluciones verdaderas*".

Al conjunto de datos tomados como soluciones verdades se los conoce como **conjunto de entrenamiento**.

Notación
- m : Cantidad de ejemplos de entrenamiento.
- x : Variable/s de entrada
- y : Variable de salida / objetivo
- (x,y) : Ejemplo de entrenamiento específico
- $(x^{(i)},y^{(i)})$: Ejemplo de entrenamiento i-ésimo



El objetivo del algoritmo de aprendizaje es tomar el conjunto de entrenamiento y generar la **función de hipótesis (h)** el cual tomará la variable de entrada (x) y *estimará* un valor a la salida (y).

Usualmente a $\mathbf{h_\theta(x)}$ se lo representa como una recta: $h_\theta(x) = \theta_0 + \theta_1 x$

Al ser una **regresión lineal**, la relación también lo es, por lo que se linealiza la transferencia.

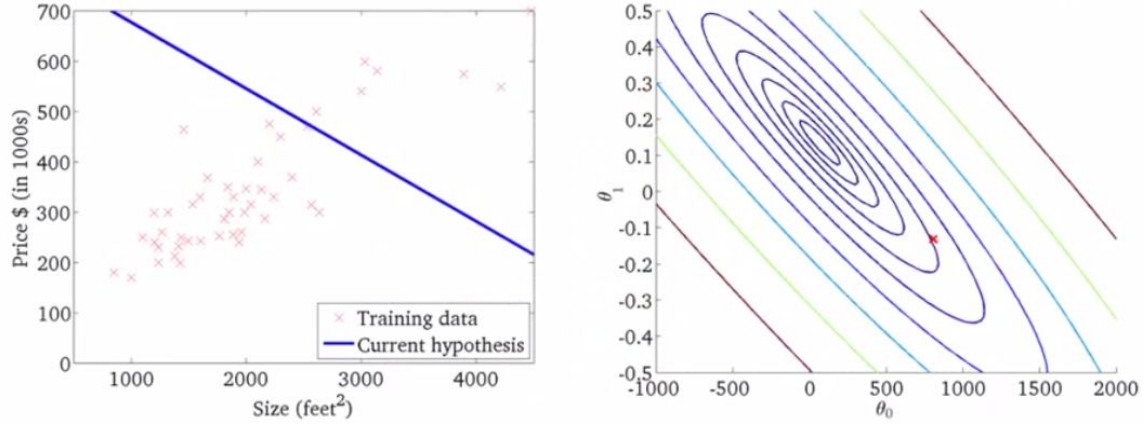
Función de coste (J): Permite ajustar la mejor línea recta posible determinando los parámetros del modelo.
- θ_i : Parámetro del modelo (θ_0 y θ_1)

Para lograr esto voy a buscar los valores de θ_0 y θ_1 que minimizan la recta, que es lo mismo que minimizar el cuadrado de la diferencia entre la hipótesis y la variable de salida.

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \qquad \underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$$

Donde $h_\theta(x^{(i)})$ es la predicción de la hipótesis evaluada en la variable de entrada "i", y de la resta con el valor real $y^{(i)}$ se obtiene un error en la estimación.
Esto se evalúa para todo el conjunto de entrenamiento (m).
A esta ecuación se la conoce como **error cuadrático**.

Ejemplo



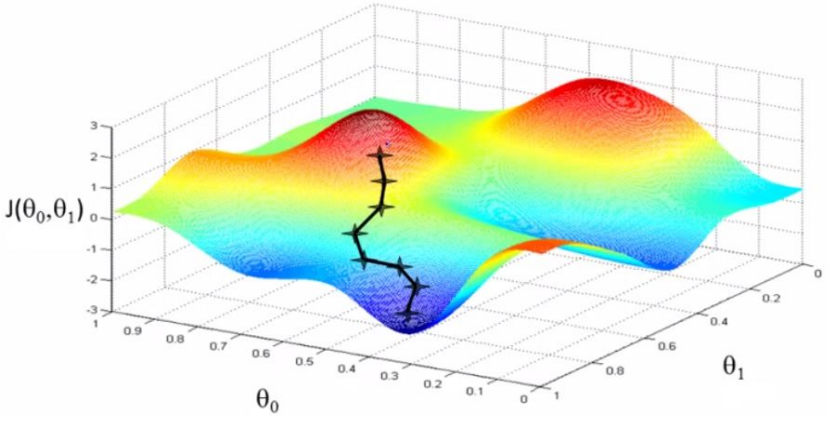
Se plantea una hipótesis y se representa en el primer gráfico como la línea azul. Luego se calcula para cada *ejemplo de entrenamiento* su *error cuadrático medio* y se lo muestra como un **gráfico de contorno** ya que es dependiente de dos variables (θ_0 y θ_1 - los parámetros del modelo).
La figura 2 muestra la función de coste para todos los posibles casos de θ_0 y θ_1, para poder minimizarla y encontrar la que me proporcione el menor error.

Método del Gradiente Descendente

Este algoritmo minimiza la función de coste "J" y no se aplica únicamente en regresión lineal.

Algoritmo

- Tomar un par (θ_0 , θ_1). *Por ejemplo θ_0=0 , θ_1=0*.
- Cambiarlos hasta reducir la función de coste J(θ_0 , θ_1) hasta encontrar el *mínimo local*.



Lo que se busca es la dirección donde el gradiente de decrecimiento sea el máximo posible.

En este ejemplo, partió del punto rojo hasta finalizar en el azul.

Si hubiera elegido un θ_1 un poco más pequeño, mi resultado

hubiera sido el otro mínimo local, por lo que puede deducirse que el resultado depende de la condición inicial.

$$\text{repeat until convergence } \{$$
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$
$$\}$$

$$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$
$$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$
$$\theta_0 := \text{temp0}$$
$$\theta_1 := \text{temp1}$$

Lo que hace el algoritmo es actualizar θ_0 y θ_1 a través de la sustracción del gradiente de la función de coste simultáneamente para ambos parámetros.

- **α : Tasa de aprendizaje**. Controla cuánto es el "paso" del algoritmo en cada instante de tiempo.

Cuánto más cerca se encuentre del mínimo, menor será el valor de la derivada por lo que el término que se resta también lo será obteniéndose "pasos" *cada vez más chicos* que evitarán una posible *divergencia* y además tener que hacer un "α" variable dinámicamente.
Cuando el algoritmo llega al mínimo, la derivada es cero, por lo que θ_J no cambia.

Gradiente Descendente aplicado a la función de coste

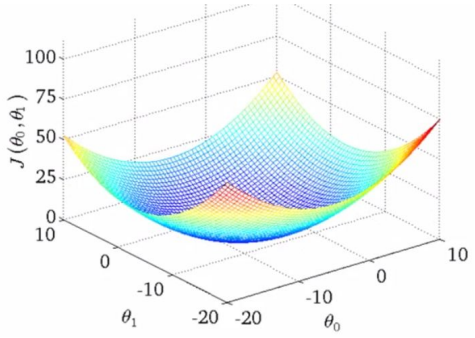
Para poder implementarlo computacionalmente debe estudiarse el término de la derivada para los dos parámetros del modelo.

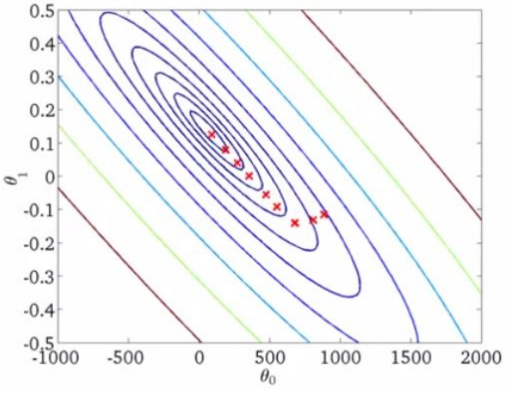
$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$
$$\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

Si reemplazo J como la sumatoria de la diferencia de cuadrados y derivo parcialmente respecto de θ_0 y θ_1 (aplicando la *regla de la cadena*) obtendré las expresiones de la izquierda que se reemplazarán en el algoritmo.

El Gradiente Descendente aplicado a las funciones de coste siempre darán como resultado un único mínimo local debido a que son **funciones convexas** como la de la figura.
Esto quiere decir que poseen un solo mínimo global.





Aplicando este algoritmo al ejemplo visto con anterioridad se observa la aproximación de la recta estimada hacia el mínimo global, obteniendo como resultado una *hipótesis* con el mínimo error posible.

Como puede observarse en el algoritmo, este método por cada instante de tiempo, *utiliza todos los ejemplos de entrenamiento* por lo que requiere mucho tiempo de procesamiento. Para evitar esto existen métodos *estocásticos* del Gradiente Descendente.
También existe otro método que evite el uso de iteraciones como lo es el **Método de las Ecuaciones Normales**.

III. Regresión Lineal con múltiples variables

Hay casos donde la cantidad de variables sea muy grande (x_1, x_2, …, x_n), por lo que cantidad de derivadas parciales volvería engorroso el cálculo. En estos casos se utilizan *matrices* y *vectores*.

Notación
- n : Cantidad de características (variables de entrada)
- $x^{(i)}$: Características del i-ésimo ejemplo de entrenamiento. $\mathbf{x^{(i)} \in \Re^n}$
- $x_J^{(i)}$: Valor de la característica "J" en el i-ésimo ejemplo de entrenamiento.

La nueva hipótesis tendrá la siguiente forma:
Para simplificar los cálculos se definirán las ecuaciones mediante *álgebra lineal*.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Se definen un vector de características y uno de los parámetros del modelo.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \cdots \\ \theta_n \end{bmatrix} \in \mathcal{R}^{n+1}$$

Reescribiendo la última ecuación, no se modifica excepto el producto interno de vectores dentro de la hipótesis: $\theta^T x$

Para obtener estas conclusiones se propuso una convención donde $x_0 = 1$.

Gradiente Descendente con múltiples variables

Ahora se pensará a los parámetros del modelo como si fueran un único vector "θ" de dimensión "n+1".

La función de coste ahora será:
Donde cualquiera de las dos expresiones es válida.

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\theta^T x^{(i)} - y^{(i)})^2$$

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(\left(\sum_{j=0}^{n} \theta_j x_j^{(i)} \right) - y^{(i)} \right)^2$$

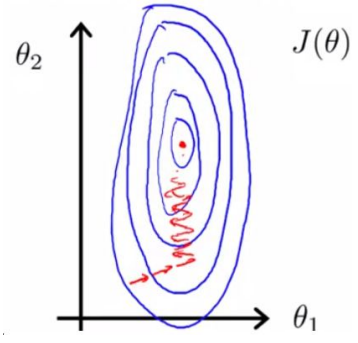
El nuevo algoritmo tendrá una forma similar una vez aplicada la derivada de J(θ).

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

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

}



Gradiente Descendente en la práctica

Feature Scalling

Se trata de que todas las variables de entrada posean una escala similar.

Ejemplo

x_1 : Tamaño (0 - 200 m^2)

x_2 : Cantidad de habitaciones (1-5)

Si se grafica la función de coste J(θ), se obtendrán elipses bastante estrechas debido a la gran desigualdad entre x_1 y x_2.

Esto ocasionará que el Gradiente Descendente tome demasiado tiempo en encontrar el mínimo global.

$$x_1 = \frac{\text{size (feet}^2\text{)}}{2000}$$
$$x_2 = \frac{\text{number of bedrooms}}{5}$$

Para solucionar esto es necesario **normalizar** (***escalar***) a las variables de entrada.

$$0 \leq x_n \leq 1$$

Mediante esta técnica los gráficos de contorno se asemejarán a círculos.

Existen casos donde es más conveniente seleccionar un rango de:

$$-1 \leq x_i \leq 1$$

Aunque casos aproximados son aceptables en la práctica.

Normalización de la media

Se reemplaza a x_i por x_i-μ_i para que las características tengan una media cero (no se aplica x_0=1).

Ejemplo

$$x_1 = \frac{size - 1000}{2000}$$
$$x_2 = \frac{\#bedrooms - 2}{5}$$

Se utiliza este método para obtener un rango de la variable nueva:

$$-0.5 \leq x_1 \leq 0.5, -0.5 \leq x_2 \leq 0.5$$

Consiste en restarle a la variable “directa” el *valor medio* ($\boldsymbol{\mu_i}$).

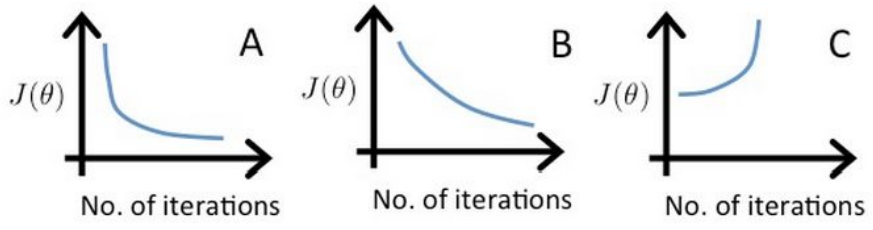
Debugging

Se utiliza para asegurarse el correcto funcionamiento del Gradiente Descendente.

Se declara convergente cuando una iteración decrece menos de $\boldsymbol{\varepsilon=10^{-3}}$, o sea que ha superado su **valor de umbral** (**threshold**).

Elección de la Tasa de aprendizaje (α)

- α demasiado pequeña : Lenta convergencia.
- α demasiado grande : J(θ) podría no decrementarse en cada iteración. No convergerá.



Es muy útil graficar **J(θ)=f(N° de Iteraciones)** para poder analizar este fenómeno.

α = (...0.001 , 0.003 , 0.01 , 0.03 , 0.1 , 0.3 , 1...)

Regresión Polinómica

Una de sus aplicaciones suele ser para *funciones no-lineales*.

Ejemplo. Predicción del precio de una casa.

Se utilizará la siguiente hipótesis y donde se han elegido las variables de entrada como:

x_1 : Longitud de la fachada

x_2 : Profundidad

$$h_\theta(x) = \theta_0 + \theta_1 \times frontage + \theta_2 \times depth$$

Pero puedo *crear* nuevas variables que me sean más provechosas, como el área del terreno:

x = frontage * depth

Entonces : $h_\theta(x) = \theta_0 + \theta_1 * x$, mi hipótesis nueva tendrá solo una variable (*área del terreno*).

Supongamos el caso de que el conjunto de datos del precio de una vivienda posee el aspecto de la figura.

Hay que encontrar el mejor modelo que se ajuste. Para ello existen dos alternativas posibles.

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

$$\begin{aligned} h_\theta(x) &= \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 \\ &= \theta_0 + \theta_1 (size) + \theta_2 (size)^2 + \theta_3 (size)^3 \end{aligned}$$

$$\begin{aligned} x_1 &= (size) \\ x_2 &= (size)^2 \\ x_3 &= (size)^3 \end{aligned}$$

Para ello se declaran las 3 variables de la siguiente manera y se escalan de acuerdo a su rango.

La otra opción posible es la de ajustar mediante una raíz cuadrada.

$$h_\theta(x) = \theta_0 + \theta_1 (size) + \theta_2 \sqrt{(size)}$$

En ambos casos del ejemplo, para casas de gran tamaño, el precio NO disminuirá (como en el caso de elegir x^2) sino que o aumentará (x^3) o se mantendrá constante (raíz cuadrada).

Ecuación Normal

Se utiliza en algunos casos particulares de regresión lineal para encontrar los valores óptimos de θ. Consiste en resolver analíticamente una ecuación de coste J(θ).

Ejemplo

$$J(\theta) = a\theta^2 + b\theta + c$$

Para minimizar esta función, se deriva respecto de θ y se iguala a cero Con este procedimiento se obtiene el valor de θ que hace mínima la función de coste.

$$\frac{\partial}{\partial \theta_j} J(\theta) = \cdots = 0 \quad \text{(for every } j\text{)}$$

Para el caso de regresión lineal con múltiples variables habría que minimizar esta función respecto a cada θ_j.

La derivación tiene la <u>desventaja</u> que es extensa y tediosa.

Ejemplo

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$$

Se escriben las variables de entrada como una matriz “X”, llamada **matriz de diseño** y adoptando todos los $\mathbf{x^{(i)}_1=1}$.

Nótese el vector columna de unos

Lo mismo hago para las variables de salida.

$$y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

El valor de θ que minimiza la función de coste se obtiene con la ecuación de la izquierda.

$$\theta = (X^T X)^{-1} X^T y$$

- Filas : ejemplos de entrenamiento $\mathbf{x}^{(i)}$.
- Columnas : variables de entrada.

En MATLAB: `Theta = pinv(X' * X) * X' * y;`

Con este método el *Feature Scalling* no es necesario.

Comparación entre métodos

Gradiente Descendente	***Ecuación Normal***
Elección de α	No es necesario elegir α
Necesita muchas ***iteraciones***	No realiza iteraciones
Funciona bien incluso con gran cantidad de características (variables de entrada)	Es necesario calcular $(X^T \cdot X)^{-1}$. **O(n^3)** Es ***lento*** para muchas variables de entrada (*n > 10.000*)

Ecuación Normal y la No-Invertibilidad

Hay ocasiones en donde la matriz $X^T \cdot X$ *no es invertible*. A este tipo de matrices se las conoce como **matriz singular**.

En MATLAB para solucionar esto se utiliza la **pseudoinversa de Moore-Penrose**, la cual siempre determinará el valor de θ incluso cuando no sea posible hallar su inversa.

Esto se puede deber a dos causas:

- Que las variables sean linealmente dependientes (L.D.) entre sí.
- O que haya muy pocos ejemplos de entrenamiento respecto a la cantidad de variables de entrada (*m* ≤ *n*). En este caso convendría:
  - Eliminar algunas características.
  - Implementar una **regularización**.

IV. Regresión Logística

Problemas de Clasificación

La variable a predecir (y) es discreta, a diferencia de los problemas de regresión. Para estos caso, la variable puede tomar dos valores.

$$y \in \{0, 1\}$$

Esto se conoce como el **Problema de Clasificación Binaria**.

En cambio, existen casos en dos la variable puede tomar más de dos valores, se conocen como **Problemas multiclase** (**y ∈ {0,1,2,3}**).

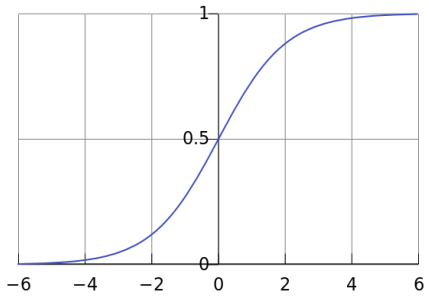
No es recomendable aplicar regresión lineal a problemas de clasificación ya que $h_\theta(x)$ puede ser mayor que 1 ó menor que 0. En cambio yo necesito que:

$$0 \leq h_\theta(x) \leq 1$$

La hipótesis debe encontrarse entre 0 y 1, ya que en ese rango estarán las variables de salida.

Para representar a esta hipótesis se utiliza la **Función Sigmoide** o *Función Logística* y se expresa matemáticamente como:

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



Y se observa en el gráfico de la derecha su representación.

Ejemplo probabilístico

- $h_\theta(x)$: probabilidad estimada de que **y=1** dada una entrada x.

Dado el siguiente ejemplo de entrenamiento, se tiene una probabilidad del 70 % de poseer un tumor maligno.

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{tumorSize} \end{bmatrix}$$
$$h_\theta(x) = 0.7$$

Probabilísticamente puede escribirse como:

$$P(y = 0|x; \theta) + P(y = 1|x; \theta) = 1$$
$$P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$$

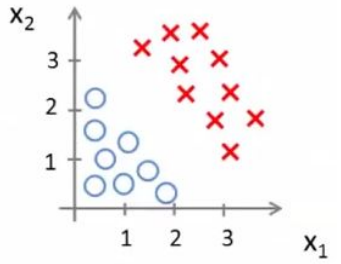
Regiones de Decisión (Decision Boundary)

Para clasificar se asume que para y = 1, $h_\theta(x) \geq 0.5$, por lo que $\boldsymbol{\theta^T \cdot x \geq 0}$.
En cambio para y = 0, $h_\theta(x) < 0.5$, y para ello $\boldsymbol{\theta^T \cdot x < 0}$.
El valor de la salida depende de si la hipótesis es mayor o menor de 0.5.

Ejemplo

Se tiene un ejemplo de entrenamiento como el de la figura.
Se usa una hipótesis de la forma:

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$



Con los siguientes parámetros:

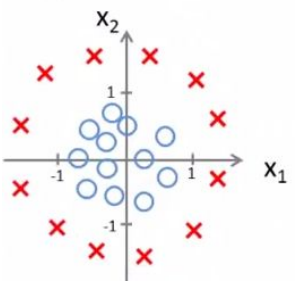
$$\theta = \begin{bmatrix} -3 & 1 & 1 \end{bmatrix}^T$$

Quiero analizar en qué zonas predecirá y=1 e y=0.

- Para y=1 : $-3 + x_1 + x_2 >= 0$, o sea, $\theta^T \cdot x \geq 0$.

Como resultado del análisis se obtiene una recta $x_1+x_2=3$ que corta a ambos ejes en el valor '3'.
A esta línea se la conoce como **Región de Decisión**.

Regiones de Decisión no-lineales



Para trazar una región no lineal debo tener una hipótesis polinómica de mayor orden. Para ello utilizo la siguiente:

$$h_\theta(x) = g\left(\theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \theta_3 \cdot x_1^2 + \theta_4 \cdot x_2^2\right)$$

Agregué dos características más a la hipótesis.
Mi vector de parámetros será:

$$\theta = \begin{bmatrix} -1 & 0 & 0 & 1 & 1 \end{bmatrix}^T$$

Con esta elección de parámetros obtengo que: $-1 + x_1^2 + x_2^2 \geq 0$

O sea que la región de decisión será una circunferencia de radio unitario centrada en el origen.

Función de Coste para Regresión Logística

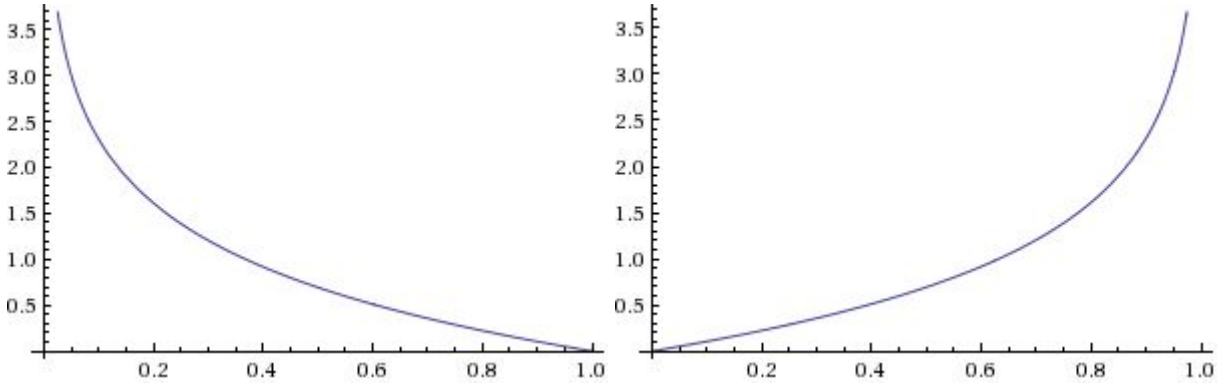
Se buscarán los parámetros θ una vez dado el ejemplo de entrenamiento.
La Función Coste utilizada para este tipo de Regresión será la siguiente:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

En estas dos gráficas puedo interpretar que para el caso donde y=1, si mi hipótesis es $h_\theta(x)=0$, entonces el costo tenderá a infinito.
O sea que es imposible que ocurra, pero en el caso de que lo haga se penalizará al algoritmo.

En cambio, el costo es mínimo (= 0) cuando para y=1, $h_\theta(x)=1$.



Función de Coste simplificada

Se reescribirá el sistema de ecuaciones para la función de coste, en una sola ecuación.

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

Esta ecuación se obtuvo al aplicar el método de **estimación por máxima verosimilitud**, el cual da una idea de como encontrar parámetros de manera eficiente.
Esta función tiene la particularidad de que es convexa, o sea, que posee un único mínimo local.

Gradiente Descendente

Habiendo derivado la función de coste para obtener el mínimo, se obtiene:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

Este algoritmo es idéntico al utilizado para Regresión lineal.
La única diferencia se encuentra en que la hipótesis es un función Sigmoide.

Vectorización

En vez de calcular, en cada iteración los "j" parámetros θ, podría implementar la siguiente instrucción usando a θ como vector.

$$\theta := \theta - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}]$$

Algoritmos de Optimización Avanzada

Se tratará de implementar el Gradiente Descendente lo más rápido posible.
Estas implementaciones son notables cuando se utilizan una gran cantidad de características.
Existen muchos algoritmos de optimización, pero para mencionar algunos:
- *Gradiente Descendente*
- *Gradiente Conjugado*
- *BFGS (Algoritmo de Broyden-Fletcher-Goldfarb-Shanno)*
- *L-BFGS (Algoritmo BFGS para memoria limitada)*

Con estos tres últimos algoritmos no es necesario calcular la tasa de aprendizaje (α) porque lo hace automáticamente haciendo que converjan más rápidamente que los métodos convencionales.

Ejemplo

Se poseen los siguientes parámetros:

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$
$$J(\theta) = (\theta_1 - 5)^2 + (\theta_2 - 5)^2$$
$$\frac{\partial}{\partial \theta_1} J(\theta) = 2(\theta_1 - 5)$$
$$\frac{\partial}{\partial \theta_2} J(\theta) = 2(\theta_2 - 5)$$

Para optimizar dichos parámetros en MATLAB podría implementarse las siguientes instrucciones:

```
function [jVal, gradient] = costFunction (theta)
      jVal = (theta(1)-5)^2 + (theta(2)-5)^2;
      gradient = zeros(2,1);
      gradient(1) = 2 * (theta(1)-5);
      gradient(2) = 2 * (theta(2)-5);
```

Y para realizar una optimización avanzada:

```
options = optimset('GradObj' , 'On' , 'MaxIter' , '100');
initialTheta = zeros(2,1);
[optTheta,functionVal,exitFlag] = fminunc(@costFunction,initialTheta,options);
```

El nombre de la función "fminunc" proviene de "*function minimization unconstrained*".
Se coloca "GradObj" en "On" ya que se proveerá un gradiente al algoritmo.
- El arroba (@) se utiliza en MATLAB para punteros a funciones.

Para que el algoritmo funcione "initialTheta" debe ser de dos o más parámetros.

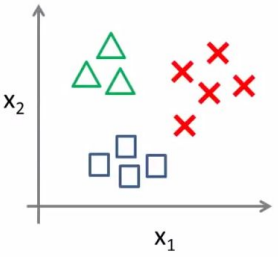
Problemas Multiclase

Las clasificaciones Multiclase se pueden utilizar, por ejemplo, para clasificar los correos electrónicos recibidos. Por ejemplo: trabajo (y=1), amigos (y=2), familia (y=3), hobby (y=4).
Otro ejemplo sería el de clasificar el clima: soleado (y=1), nublado (y=2), lluvioso (y=3).

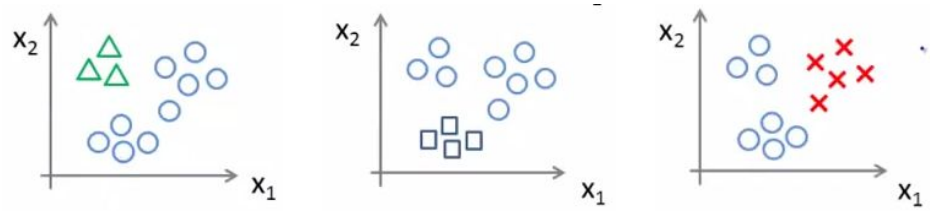
Para este tipo de problemas, los ejemplos de entrenamiento lucirán como se observa en el gráfico.

Algoritmo de Clasificación Uno Contra El Resto (One-Against-Rest)

Este algoritmo implementa los procedimientos de clasificación binaria para problemas Multiclase.



Lo que hace es subdividir al problema en *tres problemas binarios de dos clases*.
Se crean ejemplos de entrenamiento "falsos" como los que se muestran a continuación.



A las que mantienen su "forma" original se las asigna con valor positivo, mientras a las que tomar forma circular se las asigna con valor negativo.
Luego, se calcula cada hipótesis.

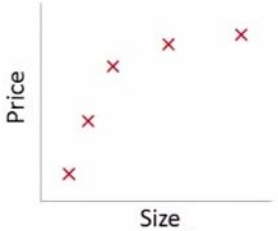
Para resumir, este método ajusta los 3 parámetros, cada uno con su hipótesis correspondiente para *estimar la probabilidad de que "y" sea igual a la clase "i".*

$$h_\theta^{(i)}(x) = P(y = i|x;\theta) \qquad (i = 1, 2, 3)$$

V. Regularización

Overfitting (Sobreajuste)

Este fenómeno puede ocasionar un rendimiento "pobre" del sistema.
Por ejemplo, en el gráfico se observa un ajuste que podría hacerse con una función cuadrática mediante un polinomio de orden 4.



$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Esto ocasiona demasiadas variaciones en la hipótesis debido a la falta de restricciones (ejemplos de entrenamiento).
O sea, que se ajusta correctamente a los parámetros de entrenamiento pero *no predice de la mejor manera las nuevas variables*.

Este problema surge en problemas de Regresión Lineal y Logística.

Para que esto no ocurre se puede proceder de la siguiente manera:
- Reducir el número de variables (características). Esto puede hacerse:
    - De forma manual
    - A través de un algoritmo de selección

Aplicando este método se está obviando información que puede ser útil para la clasificación.

- ***Regularizar***:

- Consiste en mantener todas las características pero reduce la cantidad de parámetros θ_J.
- Este método es útil cuando se tienen muchas características importantes.

Si a la función de coste la declaramos de la siguiente manera, *penalizando a θ_3 y a θ_4*, la función de orden cuarto tomará la forma de una función cuadrática.

$$\min_{\theta} \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

En general, lo que haré es modificar la función de coste agregando un **Parámetro de Regularización (λ)** a cada parámetro "θ".

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]$$

Se ha agregado un término que tenderá a achicar todos los parámetros "θ_J".
Nótese que no se penaliza a θ_0.

En esta ecuación, el primer término se utiliza para ajustar los datos a una hipótesis, mientras que el segundo término trata de mantener a los parámetros pequeños, además de mantener la hipótesis simple para evitar el Overfitting.

Aplicando esta última ecuación no se obtiene una función cuadrática exacta, pero sí una mucho más invariante.

Si utilizo un valor muy alto de λ (=10^{10}), se obtendrán valores de "θ" prácticamente nulos (≈0) con lo que se obtendrá una hipótesis $h_\theta(x)=\theta_0$. Gráficamente se representa como una línea horizontal. Esto produce un **Underfitting**.

Regresión Lineal Regularizada

Gradiente Descendente

Este algoritmo itera como en los métodos convencionales; y como θ_0 permanece invariante, tendrá la misma forma que en la Regresión Lineal no Regularizada.
En cambio, para los demás parámetros:

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

$$1 - \alpha\frac{\lambda}{m} < 1$$

Ecuación Normal

Este método agrega un término para el cálculo de la hipótesis.

$$\theta = \left(X^T X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & \ddots \\ & & & & 1 \end{bmatrix} \right)^{-1} X^T y$$

Considerando el caso particular donde el número de ejemplos es menor que el número de características: **m ≤ n**, si no estuviese el término de regularización, la matriz X^TX sería singular por lo que no se podría hallar su inversa directamente.
En cambio, con el agregado, si **λ > 0**, se puede demostrar que la matriz es invertible, por lo tanto no es necesario implementar la pseudoinversa.

Regresión Logística Regularizada

Gradiente Descendente

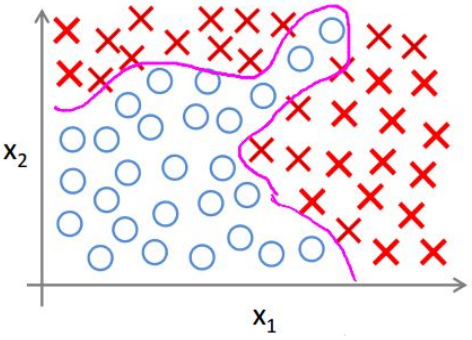
Al igual que en la Regresión Lineal, θ_0 se calcula por separado y a los demás parámetros se les agrega un término adicional.
De hecho, poseen la misma forma exceptuando que para este caso, la hipótesis es la función sigmoide.
La función coste para este tipo de regresión será:

$$J(\theta) = \left[-\frac{1}{m}\sum_{i=1}^{m} y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\cdot\sum_{j=1}^{n}\theta_j^2$$

VI. Redes Neuronales: Representación

Las **Redes Neuronales** se implementan cuando es necesario el uso de *hipótesis no lineales*.

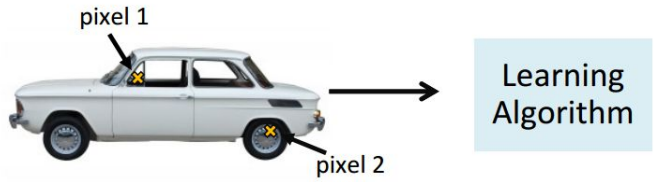
Considerando un problema de aprendizaje supervisado que tiene un ejemplo de entrenamiento como el de la figura.
Aplicar Regresión Logística de forma polinómica resolvería el caso, pero solo es útil cuando se tienen dos variables. En caso de tener más variables la cantidad de términos utilizados crecería en forma cuadrática **O(n^2)** solo si implementase a la hipótesis con términos de primer orden.

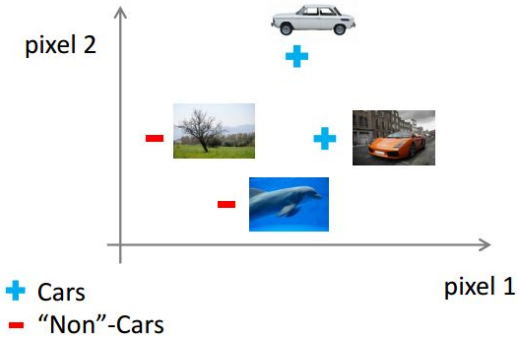


Ejemplo: Clasificación de autos usando Visión Artificial

Se parte de un conjunto de entrenamiento como el siguiente.

Lo que se hace es tomar algunos píxeles a la imagen que queremos clasificar y a las imágenes dentro del conjunto de entrenamiento.





Luego se los plotea como muestra la gráfica.

Se obtendrá que el conjunto de datos pertenecientes a "autos" estarán en una región distinta a la de "no-autos".

Para separar estas dos clases se necesitará una hipótesis no lineal.

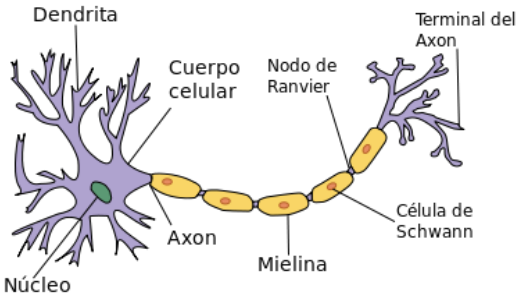
Supongamos que usamos como conjunto de entrenamiento imágenes de 50x50 píxeles, que en total hacen 2500 píxeles.
El tamaño de las variables será de n=2500 (n=7500 si son RGB).
Cada componente de x tendrá un rango de 0-255.

$$x = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

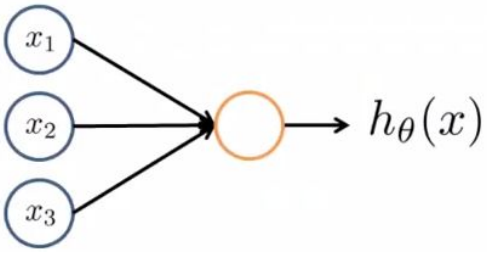
Si quisiéramos aplicar Regresión Polinómica al ejemplo, solo con términos de primer orden ($x_i \cdot x_j$) se usarían 3 millones de características.

Neuronas cerebrales

El cerebro posee cientos de neuronas.
Una parte importante de ellas es su cuerpo celular.
Cada una posee cierta cantidad de entradas (o dendritas) quienes reciben señales de otros lugares. Al terminal de salida se lo conoce como Axón.
Las neuronas se comunican entre sí a través de pulsos eléctricos.



Redes Neuronales Artificiales (ANN - RNA). Unidad Logística.

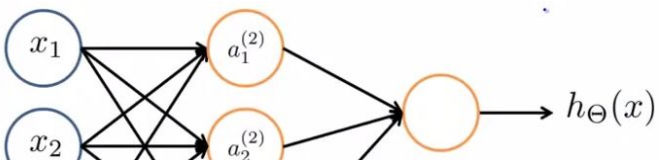


$$h_\theta(x) = \frac{1}{1 + e^{\theta^T \cdot x}}$$

- El "axón" vendría a ser la hipótesis. También se conoce como **Función de Activación Sigmoide**.
- Las "dendritas" se representan con el vector "x".
- A x_0 se lo conoce como **Neurona de Bias** (o **de Sesgo**) porque no aporta al cálculo ya que x_0=1.
- Los parámetros del modelo (θ) se conocen como los **Pesos**.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$$

Notación



- Una **Red Neuronal** es un conjunto de Neuronas interconectadas.
- En la primer capa se representan las entradas. A esta zona se la conoce como **Capa de Entrada**.
- La última se conoce como **Capa de Salida**.
- La del medio se conoce como **Capa Oculta**, y en una RNA puede haber más de una.

- $\mathbf{a}_i^{(j)}$: Activación de la unidad "i" de la capa "j".
- $\mathbf{\Theta}^{(j)}$: Matriz de pesos que controlan el mapeo entre la capa "j" y la "j+1".

$$
\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3) \\
h_\Theta(x) &= a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})
\end{aligned}
$$

En este ejemplo, la matriz de pesos será de $\mathbf{\Theta}^{(1)} \in \mathbb{R}^{3 \times 4}$. En forma general, la matriz será de orden $\mathbf{(s_{j+1}) \times (s_j+1)}$.

Vectorización

Se tomará como base el ejemplo anterior. Se implementarán las siguientes notaciones:
$a_1^{(2)}=g(\mathbf{z}_1^{(2)})$ - El superíndice indica que los valores pertenecen a la capa 2.
$a_2^{(2)}=g(\mathbf{z}_2^{(2)})$ - El supraíndice indica la neurona (1,2,3) de la capa oculta.
$a_3^{(2)}=g(\mathbf{z}_3^{(2)})$ - "**z**" es una *combinación lineal ponderada* de los valores de entrada (x_0,x_1,x_2,x_3).

Habiendo definido a "x" con anterioridad, el vector "z" será:
Y se obtiene con:

$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$z^{(2)} = \Theta^{(1)} \cdot x$ para generalizar: $x = a^{(1)}$

$z^{(2)} = \Theta^{(1)} \cdot a^{(1)}$ (3 x 4) * (4 x 1) = (3 x 1)

$a^{(2)} = g(z^{(2)})$ [a0;(3 x 1)] = (4 x 1)

Los vectores $a^{(2)}$ y $z^{(2)}$ son de dimensión $\mathbb{R}^3$, o sea que la función sigmoide se aplica a cada elemento por separado.
Se generalizó la variable x como $a^{(1)}$ para hacer más consistente el algoritmo.
Luego, debe tenerse en cuenta la neurona de bias de la capa oculta ($\mathbf{a}_0^{(2)}$) y agregarla al vector de componentes de $a^{(2)}$.
Por último, se realizan las siguientes operaciones:
Los "a" son las activaciones.

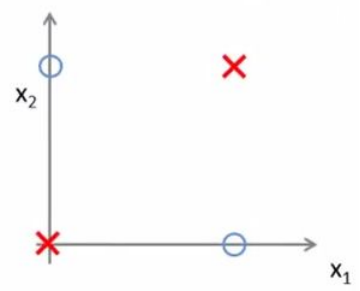
$$z^{(3)} = \Theta^{(2)}a^{(2)}$$
$$h_\Theta(x) = a^{(3)} = g(z^{(3)})$$

A este método se lo conoce como **Propagación Directa**.

Ejemplo de Clasificación No Lineal : **XOR/XNOR**

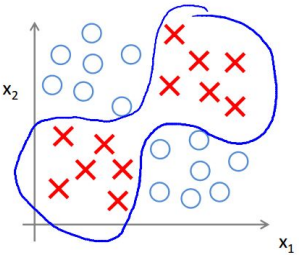
Una Red Neuronal calculará una función no lineal compleja de entrada.

Este ejemplo demostrará que las **Redes Neuronales pueden utilizarse para aprender hipótesis no lineales**.



Se posee el siguiente gráfico con variables binarias x_1 y x_2.

Este ejemplo es una versión simplificada de ejemplos más complejos como el de la figura inferior.
Puede observarse que la Región de Decisión no es lineal.



$$y = x_1 \text{ XNOR } x_2$$

Este algoritmo es válido cuando ambas variables (x1 y x2) son verdaderas o falsas simultáneamente. Similar a una compuerta XNOR.
- x : y=1
- o : y=0

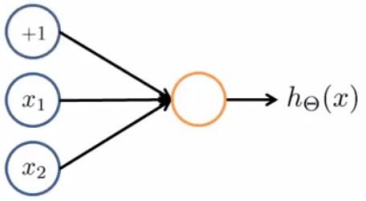
Ejemplo: Compuerta AND

Los datos del problemas son los siguientes:

$$x_1, x_2 \in \{0, 1\}$$
$$y = x_1 \text{ AND } x_2$$

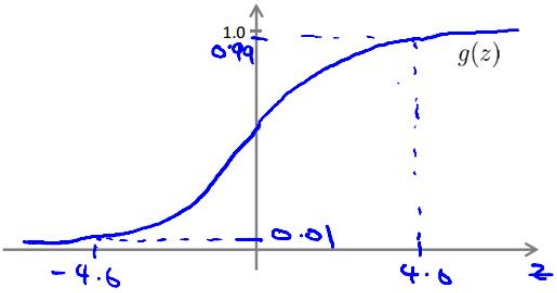
A la Red Neuronal se le agregó la neurona de bias.
La matriz de ponderación es :
Esto quiere decir que:

$$h_\Theta(x) = g(-30 + 20 \cdot x_1 + 20 \cdot x_2)$$



$$\Theta^{(1)} = \begin{bmatrix} -30 \\ +20 \\ +20 \end{bmatrix}$$

El tiempo de crecimiento (Rise Time - T_R) de la función va desde -4,6 a 4,6.



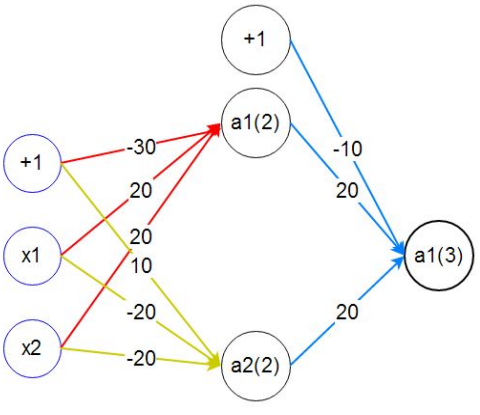
Se analizarán en la siguiente tabla los cuatro posibles valores de las entradas:

x_1	x_2	$h_\Theta(x)$
0	0	*g(-30) ≈ 0*
0	1	*g(-10) ≈ 0*
1	0	*g(-10) ≈ 0*
1	1	*g(10) ≈ 1*

Ejemplo: XNOR

Sabiendo que su expresión lógica $A \cdot B + \overline{A} \cdot \overline{B}$ es:

- $a_1^{(2)}$ es una compuerta AND mientras que $a_2^{(2)}$ es una NOR.
- El nodo de salida $a_1^{(3)}$ es una compuerta OR.



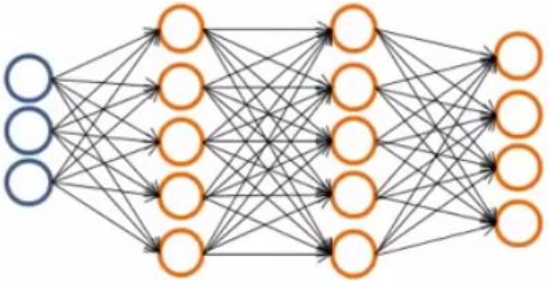
Obsérvese la diferencia entre los pesos de la OR y la NOR.

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Clasificación Multiclase

Se trata de distinguir entre varias clasificaciones.
Las Redes Neuronales son una extensión del método "***Uno contra el Resto***".
El ejemplo de entrenamiento utilizado se muestra a continuación.

La Red Neuronal utilizada posee cuatro Neuronas en la Capa de Salida, una por cada clasificación. Esto es: $h_\Theta(x) \in \mathbb{R}^4$



Por ejemplo, para el caso de los peatones:

$$h_\Theta(x) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Este es un ejemplo de cuatro Clasificadores de Regresión Logística cada uno se encarga de identificar a una clase en particular.
Los conjuntos de entrenamientos se identifican como un par: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$
Donde $\mathbf{x^{(i)}}$ es la imagen e $\mathbf{y^{(i)}}$ es la clasificación correspondiente.
La relación entre ellos es que $\mathbf{h_\Theta(x^{(i)}) = y^{(i)}}$.

VI. Redes Neuronales: Aprendizaje

Se hallará la Función de Coste para problemas de Clasificación con Redes Neuronales.
Se tiene un conjunto de entrenamiento con "m" ejemplos: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$
Se denotará:
- **L** : Número de capas de la Red Neuronal.
- $\mathbf{s_l}$: Es la cantidad de neuronas en la capa "l" (sin contar la Neurona de Bias).
  En el ejemplo superior: s_1=3, s_2=5, s_3=5, s_4=4.

Existen dos tipos de clasificaciones:
- **Clasificación Binaria**: Una sola Neurona de salida (y puede tomar valores 0 ó 1 - **K=1**).

- **Clasificación Multiclase**: Posee k-neuronas de salida.

En el ejemplo anterior, K=4 y el vector de salida:

$$h_\Theta(x) \in \mathbb{R}^K$$

Función de Coste de una Red Neuronal

Es una generalización de la Función de Coste de Regresión Logística.

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \cdot \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \cdot \log(1 - h_\Theta(x^{(i)}))_k\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(\Theta_{ji}^{(l)})^2$$

Se agrega el término de la sumatoria de las "k" neuronas.
El término de regularización no contempla las Neuronas de Bias.

Algoritmo de Retropropagación (Backpropagation)

Este algoritmo se encargará de minimizar la Función de Coste, buscando los valores óptimos de Θ.
Como en los métodos anteriores, hay que buscar las derivadas parciales $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ de

Este algoritmo se encarga de calcular el error del nodo "j" en la capa "l" y se lo asignará a $\delta_j^{(l)}$:

Como ejemplo: $\delta_j^{(4)} = a_j^{(4)} - y_j$ Donde $a_j^{(4)} = h_\Theta(x)_j$

Luego de calcular el error de la última capa, se procede con las capas interiores.

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} .* g'(z^{(3)})$$
$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

Donde el producto **.*** es el mismo que se utiliza en MATLAB.
El elemento **g'(z$^{(l)}$)** es la derivada parcial de la función de activación g(z) evaluada en los valores de entrada dados por z$^{(l)}$.

Todo este término puede escribirse como: $g'(z^{(l)}) = a_j^{(l)} .* \left(1 - a_j^{(l)}\right)$

No existe error en la primer capa (**δ$^{(1)}$**) porque son los ejemplos de entrenamiento.
La derivada de la Función de Coste (para el caso donde **λ=0**) tendrá la forma:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \cdot \delta_i^{(l+1)}$$

Para el caso donde se tengan "m" ejemplos de entrenamiento, el algoritmo a aplicar sería el siguiente:

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j). Se setean las derivadas parciales a cero.
Se utilizan como acumuladores.

For $i = 1$ to m
  Set $a^{(1)} = x^{(i)}$
  Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
  Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$
  Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$
  $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Este lazo recorre los “m” ejemplos de entrenamiento.
El algoritmo toma el primer ejemplo y realiza una **Propagación Directa** a través de la Red Neuronal. Una vez hecho esto, se calcula el error de la Neurona final y los errores restantes mediante Retropropagación.
Luego se trabajó con el término acumulador de derivadas parciales. Esta última expresión puede escribirse en forma vectorizada como:

$$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \cdot (a^{(l)})^T$$

$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \text{ if } j \neq 0$$
$$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} \text{ if } j = 0$$

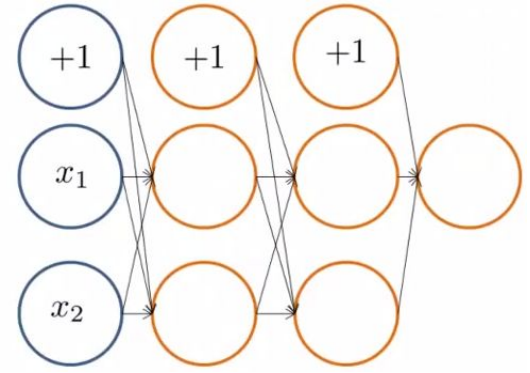
Una vez recorrido el lazo, se completa el algoritmo con las siguientes instrucciones:

Esta es una misma ecuación, pero tiene distinta resolución de acuerdo al valor de “j”.

Estos términos representan: $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Ejemplo de Backpropagation

Se analizará el algoritmo de Backpropagation con la siguiente Red, que tiene dos unidades de entrada y dos Capas Ocultas.

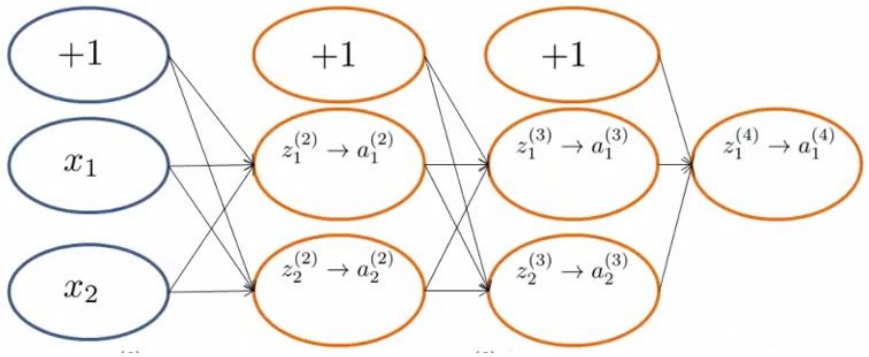


Primero se toma un ejemplo: **{$x^{(i)}$,$y^{(i)}$}**.

Luego se obtienen las funciones de activación (**$a^{(2)}$**) aplicando la función Sigmoide, como se vio anteriormente.
Se realiza la Propagación Directa en todas las Capas.

Como ejemplo se toma la primer Neurona de la Capa Oculta 2:

$$z_1^{(3)} = \Theta_{10}^{(2)} \cdot 1 + \Theta_{11}^{(2)} \cdot a_1^{(2)} + \Theta_{12}^{(2)} \cdot a_2^{(2)}$$



Los errores que se calculan dan una idea de cuánto deben cambiarse los pesos de la Red Neuronal los cuales afectan los valores intermedios(de las Capas Ocultas) para lograr en la salida $h_\Theta(x)$.

Para calcular las ponderaciones en la Retropropagación, se tomará como ejemplo la segunda Neurona de la primera Capa Oculta.

$$\delta_2^{(2)} = \Theta_{12}^{(2)} \cdot \delta_1^{(3)} + \Theta_{22}^{(2)} \cdot \delta_2^{(3)}$$

Optimización avanzada

Se implementa una Función de Coste que recibe los parámetros "*theta*" como argumento y devuelve dicha función (*jVal*) y sus derivadas (*gradient*).
Luego se procede a optimizar la Función de Coste con una función previamente explicada.

```
function [jVal, gradient] = costFunction (theta)
optTheta = fminunc(@costFunction, initialTheta, options)
```

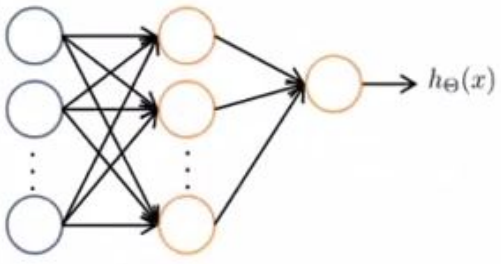
Estas dos instrucciones asumen que:
- "*theta*" y "*initialTheta*" son vectores de tamaño $\Re^{n+1}$.
- "*gradient*" es del mismo tamaño que los anteriores ($\Re^{n+1}$).

Esto funciona correctamente con Regresión Logística, pero no así con Redes Neuronales ya que :
- $\Theta^{(1)}$, $\Theta^{(2)}$ y $\Theta^{(3)}$ son matrices (*Theta1, Theta2, Theta3*).
- Los gradientes retornados ($D^{(1)}$, $D^{(2)}$ y $D^{(3)}$) también son matrices (*D1, D2, D3*).

Para que las instrucciones mencionadas previamente funcionen, deben hacerse una serie de cambios.

Ejemplo

Se tiene una Red Neuronal con la Capa de Entrada de 10 unidades, y la Capa de Salida de una única unidad. Se muestran a continuación las dimensiones de las matrices.



$$s_1 = 10, s_2 = 10, s_3 = 1$$

$$\Theta^{(1)} \in \mathbb{R}^{10\times 11}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$$

$$D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$$

En MATLAB, para convertir estas matrices en vectores se utilizan los siguientes comandos:

```
thetaVec = [Theta1(:) ; Theta2(:) ; Theta3(:)];
DVec = [D1(:) ; D2(:) ; D3(:)];
```

Con esto, se colocan todas las matrices como un vector único.
Para volver a obtener las representaciones matriciales se utiliza:

```
Theta1 = reshape(thetaVec(1:110),10,11);
Theta2 = reshape(thetaVec(111:220),10,11);
Theta3 = reshape(thetaVec(221:231),1,11);
```

Los números anteriores pueden obtenerse aplicando las funciones "*sizeof*" o "*length*".

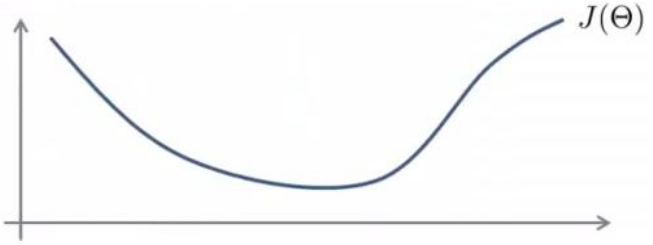
Algoritmo de Aprendizaje

- Se tienen unos parámetros iniciales $\Theta^{(1)}$, $\Theta^{(2)}$ y $\Theta^{(3)}$.
- Desenrollarlas para obtener "*initialTheta*" para ser pasada como argumento a *fminunc()*.
- Implementar *costFunction()* de la siguiente manera:
    - Obtener los parámetros iniciales con la función *reshape()*.
    - Implementar Propagación Directa e Inversa para obtener los gradientes y la Función de Coste.
    - Desenrollar los gradientes en "*gradientVec*".

Chequeo del Gradiente

Es posible que la Función de Coste disminuya con cada iteración, pero puede darse el caso que no sea la mejor estimación.

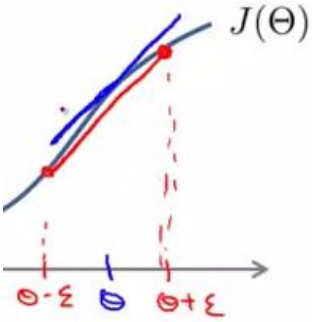
Estimación numérica del Gradiente



Se quiere estimar el Gradiente de la siguiente Función de Coste para cierto parámetro "Θ" $\in \Re$.

Se calculará la derivada algebraicamente tomando una región diferencial "ε".
Para ello se traza una recta con pendiente:

$$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta+\varepsilon)-J(\Theta-\varepsilon)}{2\cdot\varepsilon}$$



Donde el valor de $\varepsilon=10^{-4}$.
No se elige un valor más pequeño porque puede ocasionar problemas matemáticos.

En MATLAB, esta implementación se realiza como:

```
gradApprox = (J(theta+EPSILON) - J(theta-EPSILON)) / (2*EPSILON)
```

Con la cual se obtiene un buen estimador del Gradiente de la función en el punto "Θ".

Para el caso general en donde "Θ" sea un vector de dimensión "n": $\theta = \theta_1, \theta_2, \theta_3, \ldots, \theta_n$

El procedimiento consta de "n" derivadas:

$$\frac{\partial}{\partial\theta_n}J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n+\epsilon)-J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n-\epsilon)}{2\epsilon}$$

Y su correspondiente implementación en MATLAB:

```
for i = 1:n                          %"n":Cantidad de parámetros "theta"
      thetaPlus     = theta;                                  [Θ1 ⋯ Θi + ε ⋯ Θn]
      thetaPlus(i)  = thetaPlus(i) + EPSILON;
      thetaMinus    = EPSILON;                                [Θ1 ⋯ Θi − ε ⋯ Θn]
      thetaMinus(i) = thetaMinus(i) - EPSILON;
      gradApprox    = (J(thetaPlus) - J(thetaMinus)) / (2*EPSILON);
end;
```

Para asegurarme que la Retropropagación funcione correctamente, lo comparo este último resultado de modo que: $gradApprox \approx Dvec$.

La desventaja de este método es que es extremadamente lento.

Nota importante: Deshabilitar esta función cuando no sea necesaria ya que de lo contrario el código se vuelve MUY lento.

Inicialización Aleatoria

Es necesario que los parámetros (Θ) tengan valores iniciales.

Si los inicializo en cero, luego de cada actualización, las entradas que tendrá cada entrada de una capa oculta serán iguales. Esto provoca que las funciones de activación de dicha capa sean idénticas. Esto se conoce como **Problema de los Pesos Simétricos**.

Para evitar este problema se utiliza la **Rotura de la Simetría** (***Symmetry Breaking***).
El procedimiento es el siguiente:

Se inicializan aleatoriamente los parámetros ($\Theta_{ij}^{(l)}$) con valores entre [-ε ; ε]. En MATLAB:

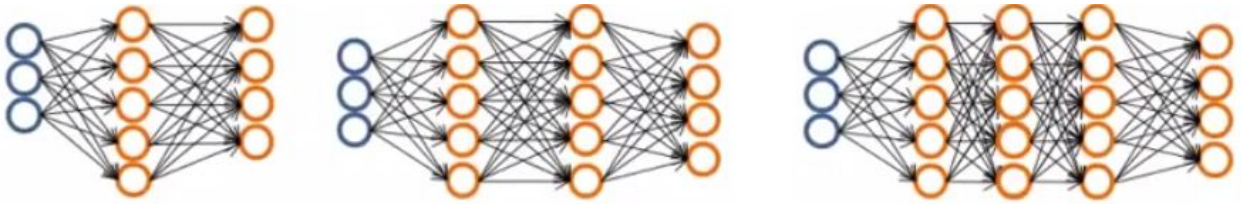
```
Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

MATLAB crea matrices del tamaño especificado con valores entre 0 y 1. Puede verse que,
- Cuando rand() = 1, Theta = ε.
- Cuando rand() = 0, Theta = -ε.

A partir de esta inicialización puede aplicarse Backpropagation para minimizar la Función de Coste.

Resúmen General

Lo primero que debe hacerse, es elegir la arquitectura de la Red Neuronal, o sea, cuál será la conexión entre las Neuronas.



La cantidad de Neuronas en la Capa de Entrada y en la de Salida estarán dadas por la información de entrada y la clasificación que quiera realizar (para una clasificación Multiclase).

Como puede verse en la figura, la cantidad de Neuronas en las Capas Ocultas es la misma.
Tener una mayor cantidad de Unidades Ocultas puede resultar útil, pero también conlleva al aumento del costo computacional.

Para entrenar una Red Neuronal deben seguirse una serie de pasos:
1) Inicializar las ponderaciones aleatoriamente.

2) Implementar la Propagación Directa para obtener $h_\Theta(x^{(i)})$ para cada $x^{(i)}$.
3) Calcular la Función de Coste J(Θ).
4) Implementar la Propagación Inversa para obtener los términos derivados de J(Θ).
Una manera sencilla de implementarlo es con una instrucción

```
for i = 1:m
```

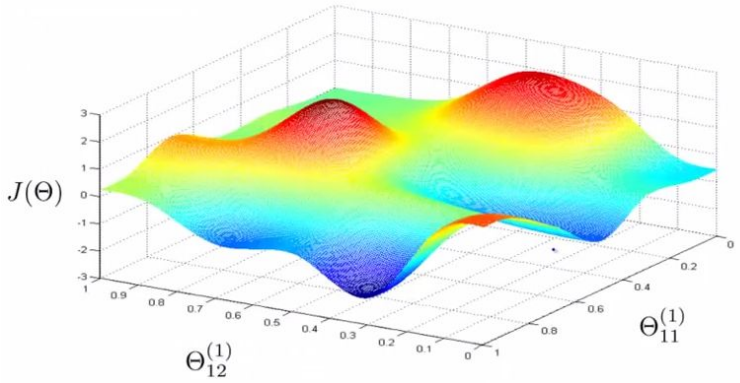
Donde “m” es la cantidad de Neuronas de Entrada (ejemplos de entrenamiento).
Con este método se obtienen las **activaciones $a^{(l)}$** y las **derivadas parciales $\delta^{(l)}$**.
Y “l” es la cantidad de Capas en la Red (**l = 2, …, L**).
Los pasos que prosiguen, son para entrenar a la Red Neuronal.
5) Usa el Chequeo del Gradiente para comparar las derivadas parciales entre los métodos de Propagación inversa y la estimación numérica.

Luego de este proceso es imprescindible desactivar dicho chequeo para no ralentizar al proceso.

Este paso me asegura que la Propagación Inversa es correcta.
6) Utilizar un algoritmo de optimización como Gradiente Descendente o algún método de optimización avanzada para minimizar J(Θ).

Para las Redes Neuronales, la **Función de Coste no es convexa** por lo que los métodos de optimización pueden quedar “estancados” en algún mínimo local.
Este inconveniente surge de manera teórica pero en la práctica no suele suceder con frecuencia.

La Función de Coste que se grafica a continuación permite medir cuán bien se ajustan los parámetros a los ejemplos de entrenamiento.



Las zonas donde el color es azul oscuro, J(Θ) es mínima, por lo que la hipótesis es bastante similar a los valores del entrenamiento.
En cambio, en las zonas donde J(Θ) es de color rojo, su representación no se ajusta correctamente a los ejemplos.

VII. Asesoramiento en la aplicación de Aprendizaje Automático

Depurando un algoritmo de aprendizaje

Para disminuir el error de ajuste del algoritmo pueden aplicarse algunas de las siguientes cosas:
- *Utilizar una mayor cantidad de ejemplos de entrenamiento*. Existen muchas ocasiones en donde no es de mucha utilidad esta herramienta.

- *Utilizar una menor cantidad de ejemplos de entrenamiento*. Se implementa para prevenir el sobreajuste u “overfitting”.
- *Agregar más variables*. Puede ser de utilidad agregar otro tipo de información adicional.
- *Agregar variables polinómicas*.
- *Incrementar λ*.
- *Decrementar λ*.

Existe una técnica que me permite saber cuáles de estas herramientas serán útiles para mejorar el algoritmo de aprendizaje llamada **Diagnósticos de Aprendizaje Automático**.
Con ella sabré qué parte del algoritmo no está funcionando correctamente.
Como contrapartida puede llevar un tiempo considerable su implementación.

Evaluación de una Hipótesis

Se evaluará una hipótesis que ha sido aprendida por nuestro algoritmo.

Para saber si se presenta el problema de overfitting puede graficarse $h_\theta(x) = f\ (variable)$, pero si nuestro conjunto de entrenamiento posee más de una variable, no será posible.

Uno de los métodos que se implementan es el siguiente:
- Se divide en dos partes (no exactamente idénticas - suele usarse 70 y 30 %) al conjunto de entrenamiento. A la menor parte se la usa de **Conjunto de Prueba** y dicha proporción se la debe elegir aleatoriamente, o sea que no es conveniente tomar valores que estén próximos entre sí.

Para *Regresión Lineal* se aplica el siguiente procedimiento:
- Se ajustan los parámetros *θ* para minimizar la Función de Coste, la cual se calcula con el 70 % del conjunto de entrenamiento.
- Se calcula el error del Conjunto de Prueba:

como la Función de Coste aplicada al 30 % restante del conjunto total.
Puede observarse que se calcula como el **error cuadrático medio**.

$$J_{test}(\theta) = \frac{1}{2 \cdot m_{test}} \sum_{i=1}^{m_{test}} [h_\theta \left(x_{test}^{(i)}\right) - y_{test}^{(i)}]^2$$

Se considerará que habrá **overfitting si $J_{TEST}(\theta)$ es grande respecto a $J(\theta)$**.

Para el caso donde se utilice la *Regresión Logística*:
- El procedimiento es idéntico al anterior solo que se aplica la siguiente ecuación.

$$J_{test}(\theta) = -\frac{1}{m_{test}} \sum_{i=1}^{m_{test}} y_{test}^{(i)} \log h_\theta(x_{test}^{(i)}) + (1 - y_{test}^{(i)}) \log h_\theta(x_{test}^{(i)})$$

Esta ecuación es idéntica a la vista con anterioridad a diferencia de que no se evalúa con la totalidad del Conjunto de Entrenamiento.

Para saber si existe overfitting se emplea el Error Clasificación Binaria (**0/1 Misclassification Error**).

$$Test\ Error = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err\left(h_\theta\left(x_{test}^{(i)}\right) - y_{test}^{(i)}\right)$$

Siendo la función

$$err(h_\theta(x) - y) = \left\{ \begin{array}{l} 1 \quad si \quad h_\theta(x) \geq 0.5 \ , \ y = 0 \\ \quad\ o \ si \ h_\theta(x) < 0.5 \ , \ y = 1 \\ 0 \ \forall otro\ caso \end{array} \right\} \ Error$$

Con esta ecuación obtendré la fracción de ejemplos que mi hipótesis ha sobreajustado.

Problemas de selección del modelo

Se va a definir una regla para saber qué grado del polinomio es mejor utilizar para que ajuste a los datos lo mejor posible. Además se determinará un valor correcto para el parámetro *λ*.

1. $h_\theta(x) = \theta_0 + \theta_1 x$
2. $h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
3. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3$

$\vdots$

10. $h_\theta(x) = \theta_0 + \theta_1 x + \cdots + \theta_{10} x^{10}$

Para saber qué grado del polinomio se implementará, se usará un parámetro nuevo:

- **d : Grado del Polinomio** que quiero utilizar.

En los ejemplos de la izquierda, el tercero posee un grado d=3.

Se quiere ver cuál de estas opciones se ajustará mejor a los Ejemplos de Entrenamiento nuevos. Un método para determinar esto, es calculando todas las $J_{TEST}(\theta^{(d)})$ y elegir la de menor valor, pero para ello es necesario definir un **Conjunto de Validación Cruzada**.

Con esta técnica, el Conjunto de Entrenamiento quedará dividido en tres partes (*Training:60%, CV:20%, Test:20%*) y por cada una se calcula un error J(θ).
Para el Conjunto nuevo, el error se calcula de la siguiente manera:

$$J_{cv}(\theta) = \frac{1}{2m_{cv}} \sum_{i=1}^{m_{cv}} (h_\theta(x_{cv}^{(i)}) - y_{cv}^{(i)})^2$$

La serie de pasos para determinar qué grado del polinomio es la más conveniente es:
- Optimizar a θ (obtener el error mínimo) para cada grado del polinomio *d*.
- Por cada $\theta^{(d)}$ obtenido, calcular $J_{CV}(\theta^{(d)})$ y tomar el de menor error.
- Tomar el polinomio de su correspondiente grado $h_{\theta(d)}(x)$ y calcular $J_{TEST}(\theta^{(d)})$ para saber si hay overfitting.

Sesgo vs. Varianza

Si la hipótesis no se ajusta correctamente al Conjunto de Entrenamiento puede ser por uno de estos dos motivos:
- Sesgo o Bias alto (*Subajuste*).
- Varianza (*Sobreajuste*).

Para analizar la diferencia entre estos dos problemas, se estudiará cómo varían los errores de entrenamiento $J(\theta)$ y de Validación Cruzada $J_{CV}(\theta)$ con la cantidad de términos polinómicos d.
- $J(\theta)$ será máxima para d=1 y mientras el polinomio sea de mayor grado, el error irá decreciendo hasta hacerse mínimo. Esta función tiene la forma de una exponencial decreciente.
- $J_{CV}(\theta)$ tendrá el mínimo error para el orden del polinomio que mejor se ajusta al Conjunto de Entrenamiento. A medida que se eligen grados más alejados al óptimo, este error irá aumentando.

Como puede observarse en el gráfico, en la zona con polinomios de bajo orden, se produce el efecto de *Underfitting*, por lo que existe un **Error de Sesgo**.
Los errores mencionados anteriormente serán altos.

$$J_{CV}(\Theta) \approx J_{TRAIN}(\Theta)$$

En cambio cuando el orden del polinomio es alto, se produce un *Overfitting* dando lugar al **Error de Varianza**.

$$J_{CV}(\Theta) >> J_{TRAIN}(\Theta)$$

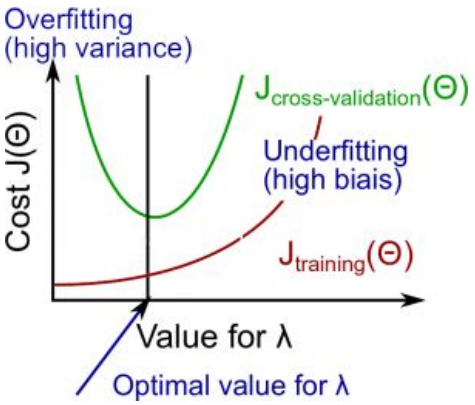
Efecto de la Regularización en los Errores

La regularización es utilizada para prevenir el *Overfitting*.

Para analizar cómo influye en los errores, se estudiarán los posibles casos del Parámetro de Regularización $\boldsymbol{\lambda}$.
- *λ grande*: Se castiga en exceso a los parámetros grandes.

$$h_\theta(x) \approx \theta_0$$

- *λ pequeño*: Se produce el Sobreajuste. Se comporta como si no se regularizara al polinomio.



Es de vital importancia notar que para este caso,

$$J_{TRAIN}(\Theta) \neq J(\Theta)$$

ya que en el segundo no se tiene en cuenta el término de Regularización.

En el gráfico se observa que:
- $J_{TRAIN}(\theta)$ es mínimo con el parámetro λ bajo ya que es donde se produce el Sobreajuste (no se penaliza a θ). En cambio, si λ es excesivamente alto, el ajuste será pobre pudiendo llegar a obtenerse hipótesis horizontales.
- Nótese que respecto a la Función de Coste sin regularizar, **los errores se encuentran invertidos**, o sea que afectan de manera contraria a los ajustes.

Elección del Parámetro de Regularización λ

Se deben seguir los siguientes pasos:

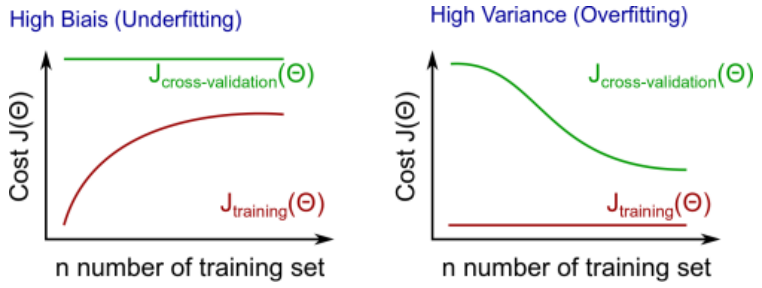
- Crear una lista con parámetros fijos λ ∈ {0, 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24}.
- Minimizar los parámetros $\Theta^{(i)}$ de acuerdo a cada λ.
- Calcular el $J_{CV}(\theta^{(i)})$ para cada parámetro. Como se observa en el último gráfico, el error mínimo será el λ óptimo.
- Calcular $J_{TEST}(\theta^{(i)})$ para verificar que no se produzca Overfitting.

Curvas de Aprendizaje

Se utilizan para saber si el algoritmo de aprendizaje funciona correctamente o para mejorar el rendimiento del mismo. Además, se puede utilizar como herramienta para detectar errores.

Lo que se hace, es graficar $J_{TRAIN}(\theta)$ y $J_{CV}(\theta)$ en función de la cantidad de ejemplos de entrenamiento *m* (o *n*).

Para el primer caso (figura de la izquierda), con pocos *m* el error de entrenamiento va a ser pequeño ya que es relativamente sencillo ajustar los parámetros a los ejemplos. A medida que la cantidad de ejemplos aumentan, resulta cada vez más complicado ajustar la curva por lo que aumenta el error.
Con el error por Validación Cruzada (figura de la derecha), sucederá lo contrario, ya que con pocos ejemplos no se podrá corroborar correctamente el funcionamiento del algoritmo.



Cuando el algoritmo subajusta los parámetros, el $J_{CV}(\theta)$ permanecerá invariante con *m* ya que un *Sesgo* alto hace que no le afecten la cantidad de ejemplos.
Como se vio previamente, para altos valores de *m*, $\mathbf{J_{CV}(\theta) \approx J_{TRAIN}(\theta)}$. Ambos poseen un alto error.
Es importante remarcar que *si el algoritmo sufre de Error de Sesgo, aumentar la cantidad de Ejemplos de entrenamiento, no reducirá dicho error*.

Cuando el algoritmo sobreajusta los parámetros, el $J_{TRAIN}(\theta)$ incrementará con *m* ya que es más complicado ajustarse debido a la cantidad de ejemplos.
Cuando se produce este tipo de fenómeno, el $J_{CV}(\theta)$ permanece en un valor alto independientemente de la cantidad de ejemplos.
Es identificable debido a la gran diferencia entre ambos errores.

Debuggeando un algoritmo de aprendizaje

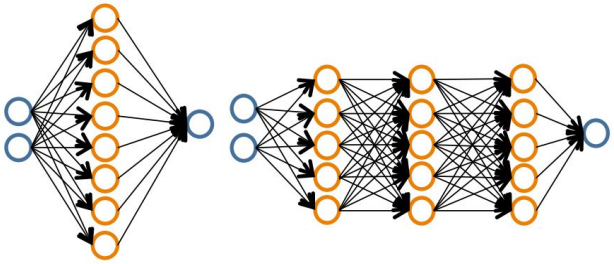
Modificando algunos parámetros se pueden solucionar errores.

- Agregar ejemplo de entrenamiento: disminuye el Error de Varianza.
- Utilizar menos variables de entrada: disminuye el Error de Varianza.

- Agregar variables de entrada: disminuye el Error de Sesgo.
- Agregar variables polinómicas: disminuye el Error de Sesgo.
- Disminuyendo λ: disminuye el Error de Sesgo.
- Aumentando λ: disminuye el Error de Varianza.

Errores en Redes Neuronales

Las redes pequeñas son más propensas a subajustar los parámetros.



En cambio, las redes grandes tienden a sobreajustar los parámetros además de ser computacionalmente más complejas.

A estas redes es conveniente regularizarlas (λ).

La cantidad de Capas Ocultas dependerá del análisis del $J_{CV}(\theta)$.

VIII. Diseño de sistemas de Aprendizaje Automático

Ejemplo de clasificación de Spam

Se comienza con Ejemplos de Entrenamiento de dos grupos:
- Correo basura (Spam), denotado como (1)
- Correo común, denotado como (0)

Se utilizará para su identificación el **Aprendizaje Supervisado**, siendo:
- *x* : características o variables de entrada del correo.

    Se eligen 100 palabras indicadoras de ser, o no, spam. Ej.: *deal*, *discount*, *buy*, (mi nombre), etc.

```
From: cheapsales@buystufffromme.com
To: ang@cs.stanford.edu
Subject: Buy now!

Deal of the week! Buy now!
```

Dado el anterior correo, puedo detectar si es spam de la siguiente manera:
- Coloco las características en una lista (pueden ser en orden alfabético).
- Creo un vector donde se pone un '1' si aparece en el correo o un 0 en caso contrario.

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \\ \vdots \end{bmatrix} \begin{matrix} \text{(mi nombre)} \\ \text{Buy} \\ \text{Deal} \\ \text{Discount} \\ \\ \text{Now} \\ \end{matrix}$$

- y : spam (1) - no spam (0)

En la práctica, no se toman palabras al azar sino las *n* palabras (10.000 a 50.000) que con mayor frecuencia aparecen.

Otra técnica para detectar spam es analizando los encabezados de los mails.

Análisis de errores

Analizar los errores me permitirá saber cómo encarar el desarrollo de los algoritmos de aprendizaje. Un procedimiento recomendado puede ser el siguiente:

- Comenzar con un algoritmo simple e implementarlo con la información de Validación Cruzada.

- Luego es conveniente dibujar las Curvas de Aprendizaje para saber si es necesario agregar mayor cantidad de ejemplos, variables, etc. Esta parte es la más complicada a la hora de desarrollar un algoritmo de aprendizaje ya que no se sabe con certeza que puede ser útil o innecesario para el mismo. A esta etapa se la conoce como **Optimización Prematura**.

- Luego se realiza el **Análisis de los errores**, que consiste en tomar los Ejemplos de Validación Cruzada y ver manualmente en qué se confunde el algoritmo; cuáles son los errores de clasificación que comete. Con este método puede detectarse un "*patrón de equivocación*".

Ejemplo

Se quiere implementar un clasificador de spam y se cuenta con 500 muestras (m_{CV}) como ejemplos de Validación Cruzada.
Observamos que el algoritmo se equivoca en clasificar 100 emails, por lo que manualmente debemos hacer:

- Clasificar el correo (*venta de drogas*, *de relojes truchos*, *phishing*, etc).
- Identificar que variables (pistas) me sirvieron para poder clasificarlo correctamente.
    - Por ejemplo, la detección de palabras mal escritas: "*m0rgage"*, "*med1cine"*, etc., que se crean con la intención de no ser detectadas por el algoritmos.
    - La dirección de email de la cual recibo el correo.
    - Utilizar demasiados signos de puntuación.

Evaluación numérica del algoritmo

Es conveniente implementar un código que nos indique cuán acertado es nuestro algoritmo.
Para mejorar su rendimiento es importante utilizar un software que detecte palabras de similar naturaleza: *discount/discounts/discounted/discounting*. A esta herramienta se la conoce como **Stemming** (Ej.: ***Algoritmo de Porter***).

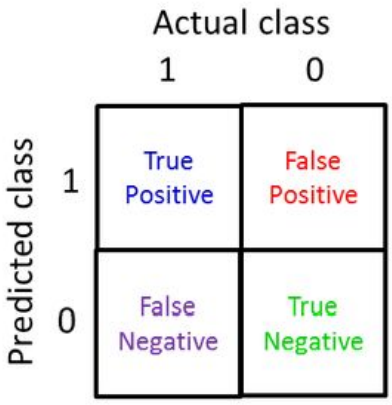
Esta implementación puede reducir el error 2%.

Métrica de errores para clases sesgadas

Ejemplo para el diagnóstico de cáncer

Se entrena un modelo de regresión logística para la detección de cáncer.
Se tiene un conjunto de entrenamiento con ejemplos positivos (*y=1*) mucho menores que ejemplos negativos (*y=0*) (**Clases Sesgadas**), por lo que resulta difícil saber si el algoritmo está ajustando bien los datos, ya que puede darse el caso que el error disminuya porque se están prediciendo resultados negativos $h_\theta(x) < 0,5$ erróneos que hacen aumentar la precisión.

Predecir constantemente resultados negativos (o positivos, dependiendo del caso) no parece ser un buen clasificador.



Para solucionar este inconveniente pueden utilizarse un método conocido como **Precisión y Exhaustividad** (**Precision and Recall**).

La terminología utilizada se presenta en el cuadro de la izquierda.

Se prueba el algoritmo con el Conjunto de Prueba (valores binarios: 1, 0).

Me permitirá evaluar el rendimiento del algoritmo.

Este método se divide en dos partes.

- **Precisión**: de los pacientes que fueron pronosticados con cáncer ($h_\theta(x) \geq 0,5$) se desea saber qué cantidad de pacientes poseen en realidad cáncer ($y=1$).

$$\frac{\text{True Positives}}{\text{Total number of predicted positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False positives}}$$

Se utilizan los valores de la primer fila de la tabla.

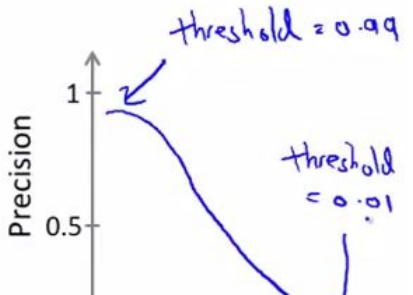
- **Exhaustividad**: de los pacientes que en realidad tienen cáncer ($y=1$), se desea saber qué porcentaje fueron predichos correctamente ($h_\theta(x) \geq 0,5$).

$$\frac{\text{True Positives}}{\text{Number of actual positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False negatives}}$$

Se utiliza la primer columna de la tabla.

Compensación de la Precisión y la Exhaustividad

Se entrenará un clasificador de Regresión Logística para que: $0 \leq h_\theta(x) \leq 1$.
- Predice cáncer si $h_\theta(x) \geq 0,5$.



- No predice cáncer si $h_\theta(x) < 0,5$.

Pero puede darse la posibilidad de que quiera predecir positivamente ($y = 1$) sólo si se tiene alta certeza del hecho.
Para tener en cuenta esta suposición se debe cambiar el umbral de decisión.
- Predice cáncer si $h_\theta(x) \geq 0,7$.
- No predice cáncer si $h_\theta(x) < 0,7$.

Esto generará que el clasificador tenga alta precisión, pero baja exhaustividad.

De acuerdo al valor de umbral que se elija, la precisión y la exhaustividad pueden variar como se observa en la figura.

Ajuste automático del umbral: F_1 Score (Valor-F)

Se quiere decidir cuál de los tres algoritmos es mejor a partir de la selección del *threshold*.

Para ello se implementa el método de Precisión y Exhaustividad y se busca una alternativa para obtener una única solución.

	Precision(P)	Recall (R)
Algorithm 1	0.5	0.4
Algorithm 2	0.7	0.1
Algorithm 3	0.02	1.0

- Promedio: No es muy útil para Clases Sesgadas.

En cambio se utiliza **F1 Score** y los valores oscilan entre 0 y 1.

$$Promedio: \frac{P+R}{2}$$

$$F_1 Score: 2 \cdot \frac{P \cdot R}{P+R}$$

	Precision(P)	Recall (R)	F_1 Score
Algorithm 1	0.5	0.4	0.444
Algorithm 2	0.7	0.1	0.175
Algorithm 3	0.02	1.0	0.0392

Debe utilizarse el algoritmo que posee el mayor *Valor-F*.

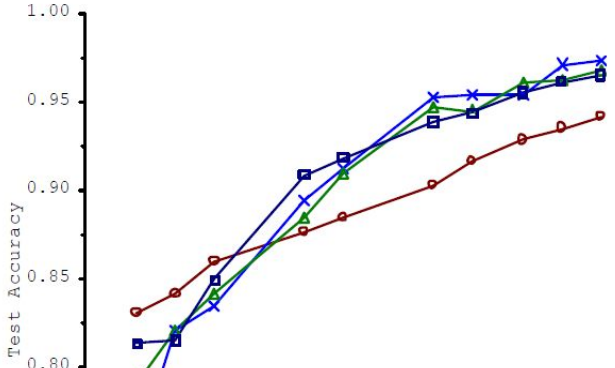
Información para el aprendizaje automático

Es importante saber cuánta información es necesaria entrenar.

[Banko,Brill] 2001
Estas dos personas investigaron el efecto de usar diferentes algoritmos de aprendizaje respecto a distintos Conjuntos de Prueba. Para ello utilizaron palabras confusas como {to,two,too}.

Los algoritmos de aprendizaje estaban destinados a categorizan oraciones donde la palabra se corresponda apropiadamente.

For breakfast I ate _two_ eggs.



Los algoritmos utilizados para la prueba fueron:
- Perceptrón (Regresión Logística)
- Winnow
- Basado en memoria
- Bayesiano Ingenuo (Naïve Bayes)

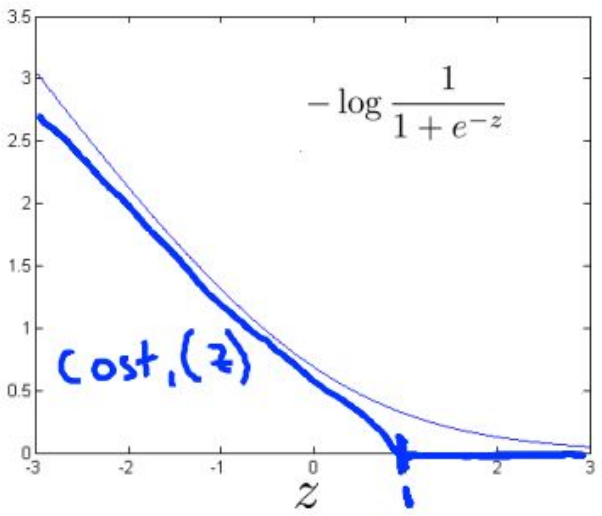
Demostraron que al aumentar la cantidad de información poseída por el algoritmo, su rendimiento mejoraba notablemente, incluso a la par de otro algoritmo que era superior a él.

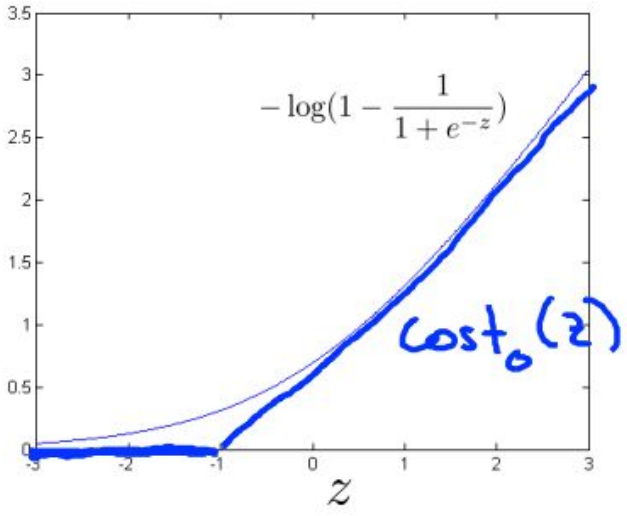
Esta cantidad de Ejemplos de Entrenamiento puede provocar un Sobreajuste cuando las variables *X* no proveen información suficiente como para predecir adecuadamente a *y*.
Y además si se utilizan algoritmos de aprendizaje simples, como Regresión Logística o Redes Neuronales con gran cantidad de Capas Ocultas.

IX. Máquinas de Soporte Vectorial (SVM)

En ciertos casos, las SVM obtienen mejores resultados que la Regresión o NN para funciones no lineales.

Las **Máquinas de Soporte Vectorial (*SVM*)** son otro tipo de algoritmo de aprendizaje supervisado.
Se modifica la Función de Coste de la Regresión Logística.
El gráfico de la izquierda se da cuando *y=1* y el de la derecha cuando *y=0*. Este tipo de función se conoce como **Hinge Loss**.





cost1(z)=max(0,k(1−z)) ***cost0(z)=max(0,k(1+z))***

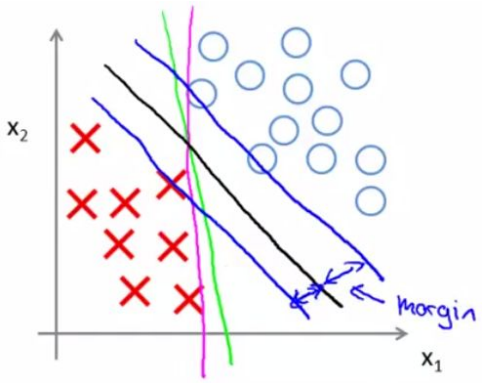
Matemáticamente la hipótesis puede expresarse como: $$h_\theta(x) = \begin{cases} 1 & \text{if } \Theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Reemplazando en la función de coste original se obtiene que:

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2$$

Se obvia la constante "*m*" ya que no aporta al cálculo del error.
Y el parámetro que controla el sobreajuste es "***C***" de la misma manera que lo hacía "*λ*".

Región de Decisión



Los SVM también son llamados **Clasificadores de Margen Máximo** ya que es posible determinar una región como la marcada en negro que posee un margen mayor (azul) que las de color.
Esto se debe a la optimización implementada.

Cuando la información puede ser separada mediante una recta, se la conoce como **Linealmente Separable**.

Kernels

Son herramientas que permiten desarrollar clasificadores complejos no lineales.
El procedimiento para implementarlo es el siguiente:
- Se tienen como dato las características "*x*".
- Se calcularán los nuevos parámetros a partir de **Puntos de Referencia $l^{(i)}$**.
- Se eligen $l^{(i)}=x^{(i)}$ y se calcula la *similaridad* entre x y cada $l^{(i)}$.

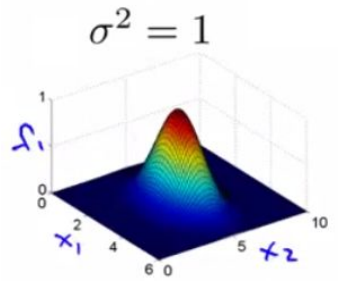
$$f_i = similarity(x, l^{(i)}) = exp(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(i)})^2}{2\sigma^2})$$

Donde el numerador es la distancia euclídea entre las variables.
A este tipo de función se la conoce como **Kernel Gaussiano**.
Cuando dos elementos están próximos entre sí, $f_i \approx 1$ es muy próximo a uno.
En cambio, cuando están alejados $f_i \approx 0$.

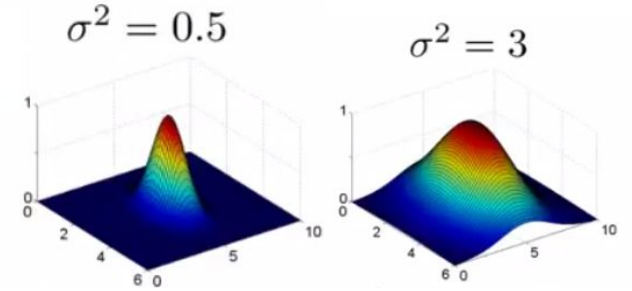
Cada Punto de Referencia $l^{(i)}$ definirá una característica f_i .

Ejemplo

$l^{(1)} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$, $f_1 = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$ Se analizará el kernel para distintos valores de desviación estándar.



Puede observarse que, cuando los valores de x_1 y x_2 son iguales a los de f_1, el valor de proximidad de máximo (=1).



En los dos ejemplos mostrados a la izquierda puede observarse cómo varía la campana de Gauss en función de la Desviación Estándar.

Nótese que a valores pequeños de δ, posee una pendiente más empinada, mientras que si posee un valor grande, la campana será menos pronunciada.

Ejemplo

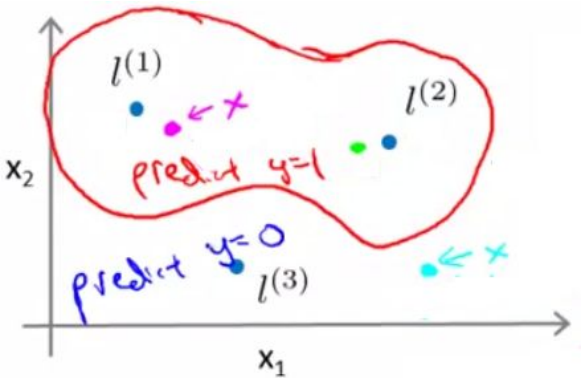
Mi hipótesis es la siguiente: $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

Los parámetros utilizados son los siguientes:
$\theta_0 = -0{,}5$; $\theta_1 = 1$; $\theta_2 = 1$; $\theta_3 = 0$

Por cada Punto de Referencia analizo su proximidad.
Por ejemplo:

$$f_1 = \theta_0 + \theta_1 \cdot 1 + \theta_2 \cdot 0 + \theta_3 \cdot 0 = -0{,}5 + 1 \cdot 1 = 0{,}5 \geq 0$$

Y por una propiedad de la SVM se predice *y=1*.



Por cada ejemplo de entrenamiento se calculan las similaridades y se las coloca en un vector.

$$x^{(i)} \rightarrow \begin{bmatrix} f_1^{(i)} = similarity(x^{(i)}, l^{(1)}) \\ f_2^{(i)} = similarity(x^{(i)}, l^{(2)}) \\ \cdots \\ f_m^{(i)} = similarity(x^{(i)}, l^{(m)}) \end{bmatrix}$$

Para obtener los parámetros "*θ*" que mejor ajustan al Conjunto de Entrenamiento, se utiliza la siguiente expresión donde puede verse que se reemplazó $x^{(i)}$ por $f^{(i)}$.

$$min_\Theta C \sum_{i=1}^{m} y^{(i)} cost_1(\Theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$$

Elección de los parámetros

$$C = \frac{1}{\lambda}$$

- Alta C : alta Varianza.
- Baja C : alto Sesgo (Bias).

- *Alto* σ^2 : alto Sesgo.
- Bajo σ^2 : alta Varianza.

Existen librerías muy buenas para optimizar el algoritmo de SVM como *liblinear* o *libsvm*.
Estos programas siguen los siguientes pasos para resolver numéricamente:
- Eligen la constante "*C*".
- Eligen el Kernel (Función de Similaridad).

- Para el caso donde se utilice el Kernel Gaussiano, es necesario definir la Desviación Estándar.

  Los Kernels son muy útiles cuando se tienen bastantes (no en exceso) Ejemplos de Entrenamiento (*m*) y poca cantidad de variables de entrada (*n*).

- También es importante realizar un "*Feature Scalling*" antes de implementar dicho Kernel.

Kernels alternativos

Para que un Kernel optimice correctamente y no diverga debe satisfacer una condición técnica conocida como el **Teorema de Mercer**.
Otros ejemplos de Kernels son:

- *Kernel Polinómico*

  $$k(x,l) = (X^T \cdot l)^2$$
  $$k(x,l) = (X^T \cdot l + 1)^3$$

  Pueden haber varias versiones, dependiendo de la constante que se le suma y del exponente.
  Se suelen utilizar cuando los parámetros de entrada son positivos.
- *String Kernel*

  Se utiliza en el caso de que la entrada sea texto.
- *Chi Cuadrada*
- *Histograma de Intersección*

X. Agrupamiento (Clustering)

Este método es un algoritmo de Aprendizaje No Supervisado que se encarga de buscar una estructura entre el Conjunto de Entrenamiento.

Training set: $\{x^{(1)}, x^{(2)}, x^{(3)}, \ldots, x^{(m)}\}$

K-medias

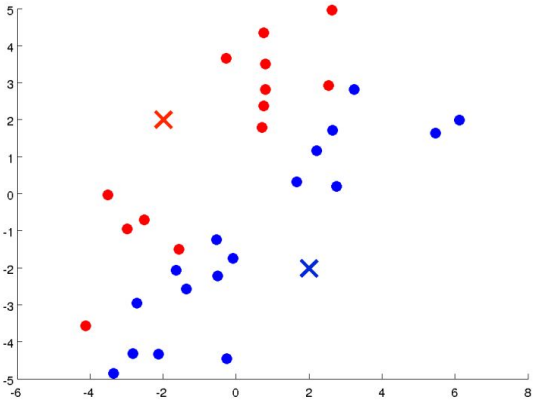
Es un tipo de algoritmo de Agrupamiento.

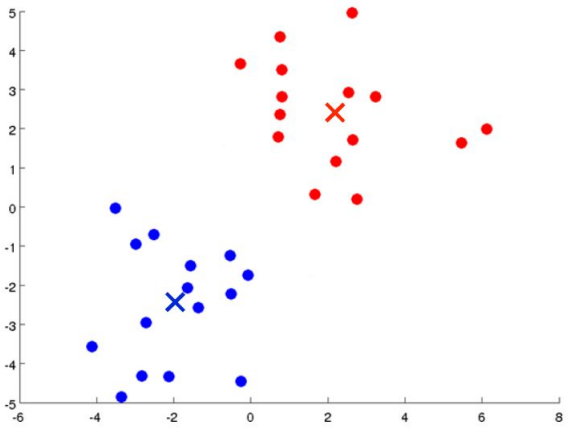
Lo primero que se hace es inicializar aleatoriamente dos puntos conocidos como **Centroides** ($\boldsymbol{\mu_K}$), debido a que quiero agrupar la información en Clústeres.

El algoritmo se desarrolla cíclicamente en dos pasos:
- Se agrupa la información en dos clústeres.
- Se mueven los centroides.

Las asignaciones en los agrupamientos se determinan de acuerdo a la mínima distancia entre ellos como se observa en la figura superior.
Para mover los Centroides, se calcula la media (el promedio). de los datos de una misma agrupación.

Como resultado de la iteración se obtiene el gráfico de la derecha.





El algoritmo utilizado es el siguiente:

Input:

- K (number of clusters)
- Training set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$

$x^{(i)} \in \mathbb{R}^n$ (drop $x_0 = 1$ convention)

Randomly initialize K cluster centroids $\mu_1, \mu_2, \ldots, \mu_K \in \mathbb{R}^n$

Repeat {

for i = 1 to m

$c^{(i)}$:= index (from 1 to K) of cluster centroid closest to $x^{(i)}$

for k = 1 to K

μ_k := average (mean) of points assigned to cluster k

}

El primer lazo, encargado del agrupamiento se puede expresar matemáticamente como:

$$c^{(i)} = \min_K \|x^{(i)} - \mu_K\|^2$$

El lazo que se encarga de mover el Centroide se expresa como:

$$\mu_K = \frac{1}{j}\sum_{i=1}^{j} x^{(i)}$$

Es importante remarcar que, si un Centroide no posee ningún punto cercano, debe eliminarse.

- **$\mu_{c(i)}$** : Centroide al cual fue asignado el ejemplo $x^{(i)}$.

La Función de Coste de este algoritmo es la siguiente:

$$\min_{\substack{c^{(1)},\ldots,c^{(m)},\\ \mu_1,\ldots,\mu_K}} J(c^{(1)},\ldots,c^{(m)},\mu_1,\ldots,\mu_K) = \frac{1}{m}\sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Se tiene que objetivo encontrar los parámetros $c^{(m)}$ y μ_k que minimizan las distancias entre los Ejemplos de Entrenamiento y los Centroides asignados a cada uno. A esta función se la conoce como **Costo de Distribución**.

Inicialización aleatoria

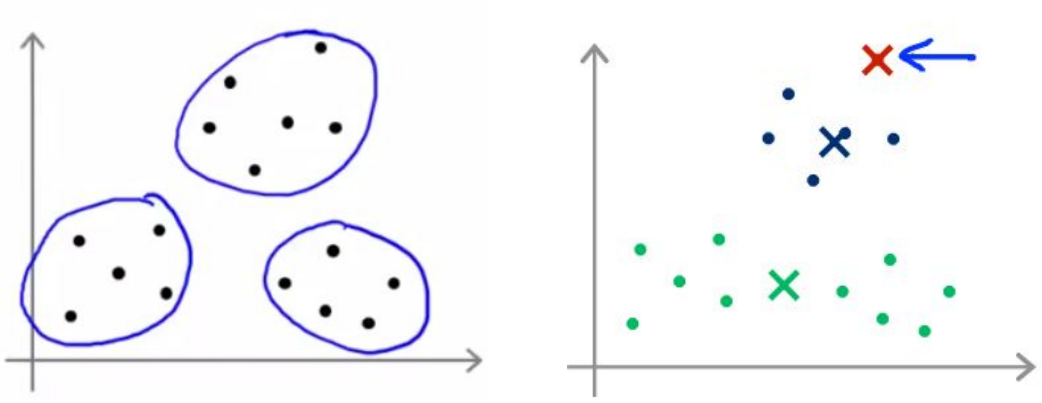
Se analizará el primer paso del algoritmo de K-medias.

Debe cumplirse que ***K < m***.

Sabiendo que debo inicializar "K" Centroides, voy a colocarlos encima de cualquier Ejemplo de Entrenamiento.

Óptimo Local

Puede darse el caso que se muestra en las figuras inferiores donde, dependiendo de la inicialización de los Centroides, puede que no se llegue a la solución correcta.
Esto quiere decir que el algoritmo finalizó en un *Óptimo Local* erróneo, y no en el *Óptimo Global*.



Para solucionar este problema se realizan múltiples inicializaciones aleatorias. El algoritmo para dicha implementación se muestra a continuación.

Se decidió correr el algoritmo unas 100 veces. Típicamente suele iterarse entre 500 y 1000 veces.

Como puede observarse, luego de realizar todas las posibles inicializaciones, se toma como correcta la que posea menor distorsión.

Este método es práctico cuando se desean realizar entre 2 y 10 agrupaciones. En cambio, cuando K > 10, con una única iteración es suficiente para encontrar el Óptimo Global.

For i = 1 to 100 {

Randomly initialize K-means.
Run K-means. Get $c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K$.
Compute cost function (distortion)
$J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$
}

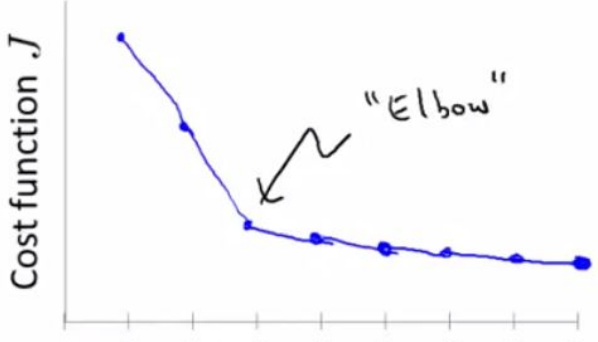
Pick clustering that gave lowest cost $J(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_K)$

Matemáticamente, este método puede expresarse como:

$$\textbf{\textit{Elegir "k" enteros aleatorios } } i_1, \ldots, i_K \textbf{\textit{ de los } } \{1, \ldots, m\} \textbf{\textit{ Ejemplos de Entrenamiento.}}$$

$$\textbf{\textit{Adoptar }} \mu_1 = x^{(i_1)}, \mu_2 = x^{(i_2)}, \ldots, \mu_K = x^{(i_K)}$$

La cantidad de Clústeres "*K*" a implementar por el algoritmo debe elegirse a mano. Para ello existen técnicas como el **Método del Codo**, donde se grafica la distorsión (*J*) en función de *K*.



Como se ve en la figura, se considera al codo donde la distorsión deja de decrementar rápidamente.

Este método no es muy utilizado porque puede resultar que la gráfica que se obtiene no tenga un codo bien definido.

XI. Reducción de la dimensionalidad

Compresión de datos

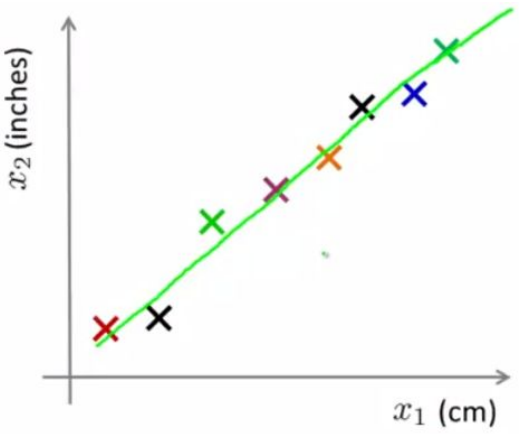
La compresión de datos me permite disminuir el costo computacional y reducir el espacio en el disco o acelerar el procesamiento de los algoritmos, entre otras cosas.

Para implementar esto, se puede disminuir la cantidad de “dimensiones” de los datos.
Por ejemplo, en el gráfico inferior se observa una reducción de 2D a 1D.

Para ello, se crea una nueva variable (*z1*) que determina la posición de las otras variables de 2D.

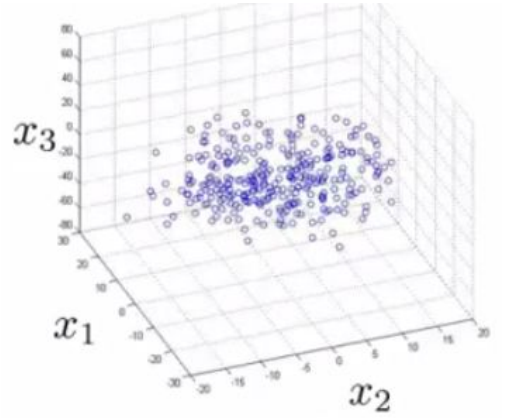
Matemáticamente esta operación se puede expresar de la siguiente manera:

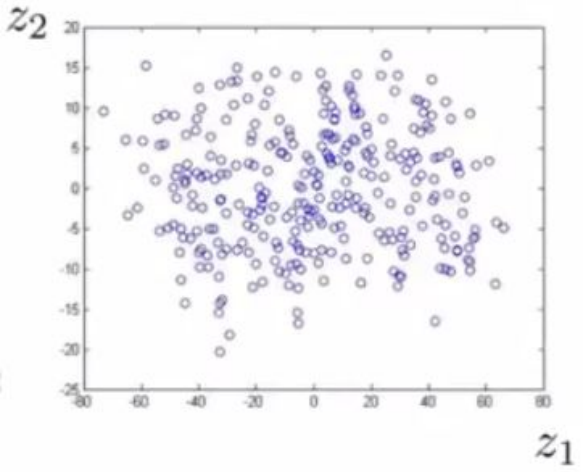
$$x^{(m)} \epsilon\, \mathbb{R}^2 \rightarrow z^{(m)} \epsilon\, \mathbb{R}$$

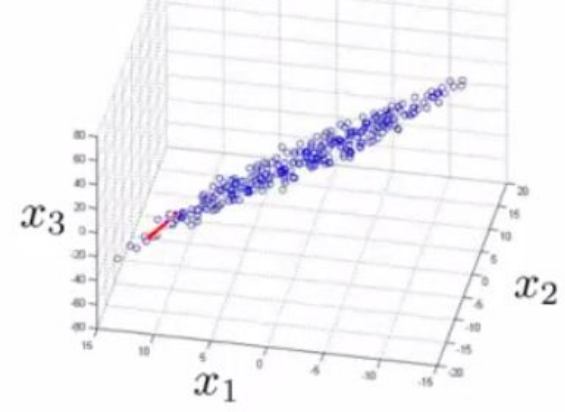


Este proceso reduce la memoria requerida a la mitad.

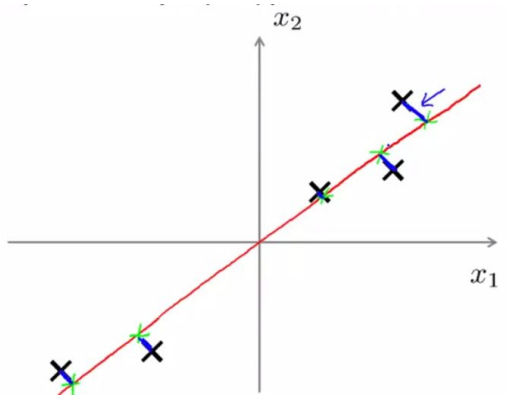
Este mismo proceso puede aplicarse a datos en 3D.
La información inicial se proyecta sobre un **plano Z_1Z_2**.







Análisis de Componentes Principales (PCA)

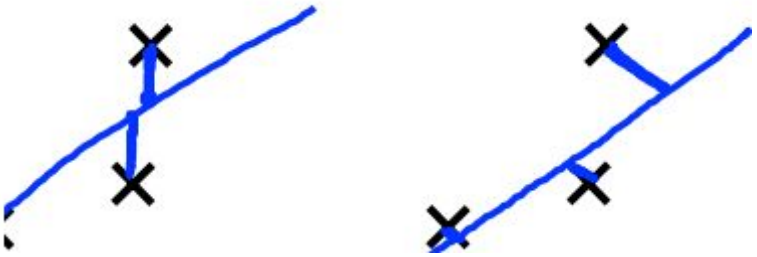


Este método consiste en determinar la mejor proyección que me permita disminuir el error al máximo (**Error de Proyección**).
Como resultado se obtiene "*k*" vectores: $\mathbf{u}^{(k)} \in \mathbb{R}^n$.

***K* : Cantidad de Componentes Principales**
Es conveniente, antes de realizar esta técnica, aplicar *Feature Scalling* y *Normalización*.

Diferencias entre PCA y Regresión Lineal

El gráfico de la izquierda muestra la Regresión Lineal y como las variables calculan el error mínimo respecto a "*y*".
En cambio, en el gráfico de la derecha, PCA calcula la proyección de las variables.



Algoritmo del PCA

Antes de comenzar el procedimiento formal del algoritmo, es necesario preprocesar la información.
Como se mencionó previamente, es necesario realizar algunas modificaciones a los datos.

Data preprocessing

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

Preprocessing (feature scaling/mean normalization):

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$$

Replace each $x_j^{(i)}$ with $x_j - \mu_j$.

If different features on different scales (e.g., x_1 = size of house, x_2 = number of bedrooms), scale features to have comparable range of values.

Luego se prosigue con la parte del algoritmo que busca los mejores vectores que logran el menor error de proyección y los valores proyectados de las variables.

Principal Component Analysis (PCA) algorithm

Reduce data from n-dimensions to k-dimensions

Compute "covariance matrix":

$$\Sigma = \frac{1}{m}\sum_{i=1}^{n}(x^{(i)})(x^{(i)})^T$$

Compute "eigenvectors" of matrix Σ:

```
[U,S,V] = svd(Sigma);
```

.

El primer paso en esta parte del algoritmo consiste en calcular la **Matriz de Covarianza** $\Sigma \in \mathbb{R}^{nxn}$ y luego los Autovalores de dicha matriz con la función *svd()* o **Descomposición en Valores Singulares**. Otra manera de calcularlos es a través de la función *eig()*, solo que la primera es numéricamente más estable.

De las tres matrices que devuelve *svd()*, la más importante es la primera y tiene la forma:

$$U = \begin{bmatrix} | & | & & | \\ u^{(1)} & u^{(2)} & \cdots & u^{(n)} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n \times n}$$

Los vectores columna que forman dicha matriz serán los que determinen el plano de proyección, por lo que deben tomarse los primeros "*K*" vectores columna. La matriz resultante será $\mathbf{U_{REDUCE}} \in \mathbb{R}^{nxK}$.

Las proyecciones entonces se calculan con la siguiente ecuación: $\boxed{z^{(i)} = U_{REDUCE}{}^T \cdot x^{(i)}}$

Donde $\mathbf{z^{(i)}} \in \mathbb{R}^K$ y $x^{(i)}$ es una parte del Ejemplo de Entrenamiento.

En MATLAB, el algoritmo PCA se implementa de la siguiente manera:

```
Sigma = X' * X / m ;
[ U , S , V ] = svd( Sigma );
Ureduce = U(: , 1:k);
z = Ureduce' * x;
```

Elección de la cantidad de Componentes Principales (*K*)

Para determinar dicho parámetro, se busca que se cumpla la siguiente relación:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \leq 0.01 \ (1\%)$$

Donde el numerador es el *Error de Proyección Cuadrático Medio*, y es lo que se desea minimizar mediante PCA. El denominador se conoce como *Variación Total de la Información*, y es la distancia de cada componente respecto al origen.

La ecuación anterior quiere decir que el 99% de la varianza es retenida.

El algoritmo se presenta a continuación.

Consiste en ir aumentando "*k*" iterativamente hasta cumplir con la condición mostrada previamente. Se suelen elegir valores máximos de *k=17*.

Esta implementación tiene un defecto, ya que necesita demasiadas iteraciones para corroborar el mejor proceso. Para evitar esto, se utiliza la siguiente instrucción.

```
[ U , S , V ] = svd( Sigma )
```

Para este caso se usa la matriz "S":

$$S = \begin{bmatrix} S_{11} & 0 & 0 & 0 \\ 0 & S_{22} & 0 & 0 \\ 0 & 0 & S_{33} & 0 \\ & & & \ddots \\ 0 & 0 & 0 & S_{nn} \end{bmatrix}$$

Algorithm:

Try PCA with $k = 1$

Compute $U_{reduce}, z^{(1)}, z^{(2)}, \ldots, z^{(m)}, x_{approx}^{(1)}, \ldots, x_{approx}^{(m)}$

Check if

$$\frac{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)} - x_{approx}^{(i)}\|^2}{\frac{1}{m}\sum_{i=1}^{m} \|x^{(i)}\|^2} \leq 0.01?$$

Este método es mucho más rápido para calcular la condición vista anteriormente ya que, para el elemento "*k*", se suman los primeros "*k*" elementos de la diagonal principal y se los divide por todos los elementos de la diagonal, como se muestra en la ecuación.

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq 0.99$$

Con este método es necesario calcular la matriz sólo una vez.

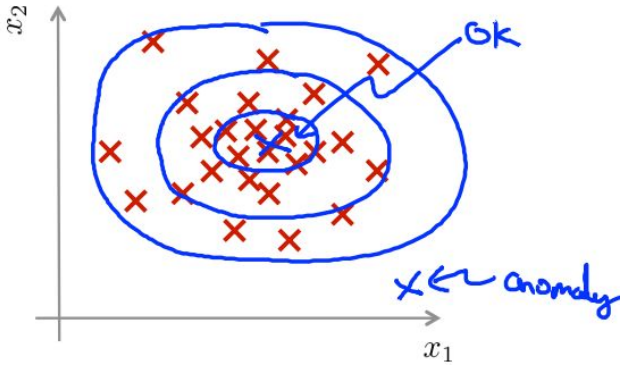
XII. Detección de anomalías

Teniendo un conjunto de datos $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ es posible decidir si un ejemplo nuevo $\mathbf{x_{test}}$ se corresponderá a dicho conjunto o será una *anomalía*.

Para determinar esto se construye un modelo ***p(x)***, que nos dice la probabilidad de un ejemplo de no ser anómalo.

$p(x_{test}) < \varepsilon$ → **Anomalía**

$p(x_{test}) \geq \varepsilon$ → **OK**



Una aplicación de esta herramienta es para *monitorear las computadoras de un data center*.

Algunas variables utilizadas serían: $x^{(i)}$ = features of machine i

x_1 = memory use, x_2 = number of disk accesses/sec,

x_3 = CPU load, x_4 = CPU load/network traffic.

El modelo se basa en una **distribución Gaussiana** de las variables. Esta ecuación se debe a que las variables son independientes entre sí

$$p(x) = \prod_{j=1}^{n} p(x_j ; \mu_j ; \sigma_j^2)$$

Algoritmo para la detección de anomalías

Choose features x_i that you think might be indicative of anomalous examples.

Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$.

Calculate $\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$

Calculate $\sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} (x_j^{(i)} - \mu_j)^2$

Given a new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} exp(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2})$$

Anomaly if $p(x) < \epsilon$

.

Los parámetros μ_j y σ_j pueden implementarse en forma vectorizada.

Ejemplo: Motores de avión

Como ejemplos se tienen:

- 10.000 motores de buena calidad (*y=0*)
- 20 motores con mal funcionamiento (anómalos) (*y=1*)

Una cantidad recomendable de ejemplos para los distintos conjuntos sería:

- Conjunto de Entrenamiento sin clasificar de 6.000 muestras (pero todas de *y=0*).
  Este conjunto se utiliza para ajustar p(x).
- Conjunto de Validación Cruzada: 2.000 con correctos y 10 con deficientes.
- Conjunto de Prueba: la misma cantidad que el CV.

Debe notarse que los subconjuntos nombrados anteriormente parten de la cantidad total de ejemplos nombrados en un principio.

Evaluación del algoritmo

- Ajustar el modelo *p(x)* al Conjunto de Entrenamiento **{$x^{(1)}$, … , $x^{(m)}$}**.
- Con el x_{CV}/ x_{TEST}, predecir:

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \end{cases}$$

- Al ser todas las muestras de y=0, es necesario realizar una “evaluación métrica”:
  - Falso positivo, verdadero negativos, etc.
  - Precisión / Exhaustividad.
  - F1
- Como complemento puede utilizarse a la x_{CV} para setear el parámetro ε.

Detección de anomalías vs. Aprendizaje Supervisado

		Detección de anomalías	Aprendizaje Supervisado
Conjunto de prueba	Positivo	*Pequeño (0-20)*	*Grande*
	Negativo	*Grande*	*Grande*
		Gran cantidad de anomalías	*Cantidad suficiente de ejemplos positivos para predecir correctamente*

Detección de anomalías con una distribución Gaussiana Multivariable

Se tienen varias anomalías x_1, x_2. Para ajustar el modelo, se utilizan al mismo momento las variables, como: *p(x)*.

Esta distribución utiliza parámetros como: $\mu \in \mathbb{R}^n$ y $\Sigma \in \mathbb{R}^{n \times n}$ (**Matriz de Covarianza**).

$$p(x;\mu,\Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-1/2(x-\mu)^T\Sigma^{-1}(x-\mu))$$

Donde:

- **|Σ|** : Determinante de la Matriz de Covarianza.
  En MATLAB se usa con : `det(Sigma)`.

Para ajustar los parámetros de dicha distribución se emplean las siguientes ecuaciones:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \quad \Sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T$$

El procedimiento del algoritmo es similar a la distribución Normal.

XIII. Sistemas de fón

Recomendaciones basadas en el contenido

Se tiene un cuadro como el que figura a continuación y se empleará una notación especial.

- $\mathbf{n_u}$: Cantidad de usuarios
- $\mathbf{n_m}$: Cantidad de películas

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

La idea de este sistema es detectar la posible puntuación de los usuarios en las películas marcadas con el signo "*?*".
Se utilizarán variables como x_1 y x_2 que medirán el grado de categorización de cada película. Sabiendo que $x_0=1$ para todos los casos, tendré que:

$$x^{(i)} = \begin{bmatrix} 0 \\ x_1 \\ x_2 \end{bmatrix}$$

Para resolver esto, se tratará a cada usuario "*j*" como un problema de Regresión Lineal. Por ende, cada uno tendrá un parámetro $\theta_j \in \mathbb{R}^3$ el cual utilizará para predecir el rating de la película "*i*" mediante:

$$\boxed{(\theta_j)^T \cdot x^{(i)}}$$

Obteniéndose como resultado la cantidad de estrellas.

Por ejemplo, para la usuaria "Alice", la película "Cute puppies of love" tendrá un puntaje de:

$$y^{(3,1)} = (\theta_1)^T \cdot x^{(3)} = \begin{bmatrix} 0 & 5 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0.99 \\ 0 \end{bmatrix} = 4.95$$

- $\mathbf{y^{(i,j)}}$: Puntuación de la película "i" según el usuario "j"
- **r(i,j)** : es 1 si el usuario ha puntuado la película. Es 0 en caso contrario.

Para minimizar el error de puntuación de todos los usuarios respecto a las películas que puntearon se utiliza, al igual que en la Regresión Lineal:

$$\boxed{min_{\theta^{(1)},\ldots,\theta^{(n_u)}} = \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left((\theta^{(j)})^T(x^{(i)}) - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2}$$

Donde el segundo término es el de regularización.
Su notación más abreviada es $\mathbf{J(\theta^{(1)},\ldots,\theta^{(nu)})}$.

Para encontrar el valor de "*θ*" que minimiza la función de coste se emplea el **Gradiente Descendente**.

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} \quad (\text{for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right) \quad (\text{for } k \neq 0)$$

Cabe recordar que el parámetro de regularización no corre para *k=0* y además, todo el término que está entre paréntesis es la derivada parcial de la función de coste respecto de *θ*.

Filtrado Colaborativo

Este filtro puede aprender por sí solo qué variables utilizar.

Supongamos ahora que se tiene el mismo conjunto de entrenamiento anterior, solo que esta vez no se conocen los valores de las variables.

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	?	?
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

A diferencia del método anterior, supondremos que los usuarios le dicen al sistema cuáles son las películas que más les gustan a través de los siguientes parámetros:

$$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(2)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(3)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}, \theta^{(4)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$$

Puede verse que a Alice y a Bob les gustan las películas románticas, mientras que a Carol y a David les gustan las películas de acción.

A partir de estos datos es posible inferior los parámetros x_1 y x_2 para cada película.

El algoritmo de optimización implementa la siguiente ecuación:

$$min_{x^{(1)},\ldots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} \left(x_k^{(i)}\right)^2$$

Y el Gradiente Descendente se calcula como:

$$x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} \left(\left(\theta^{(j)}\right)^T \left(x^{(i)}\right) - y^{(i,j)} \right) \theta_k^{(j)} + \lambda x_k^{(i)} \right)$$

Algoritmo

El procedimiento consta de dos pasos:
- Dadas $x^{(1)}, \ldots, x^{(nm)}$ se pueden estimar los parámetros $\theta^{(1)}, \ldots, \theta^{(nu)}$.
- Dados $\theta^{(1)}, \ldots, \theta^{(nu)}$ se pueden estimar las variables $x^{(1)}, \ldots, x^{(nm)}$.

Una manera de implementar este método sería repitiendo los dos pasos anteriores iterativamente, pero existe otro algoritmo más eficiente.
En realidad se implementan las dos instrucciones simultáneamente.

$$J(x,\theta) = \frac{1}{2}\sum_{(i,j):r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}(x_k^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

Donde se busca minimizar la función de coste, la cual depende de *x* y *θ* .

Si se observa las dos funciones de coste *J(x)* y *J(θ)*, remarcadas anteriormente, puede notarse que los primeros términos son casi idénticos exceptuando por los intervalos de la sumatoria.
Los otros dos términos restantes son para la regularización.

En este caso, no es necesario utilizar x_0, por lo cual $\mathbf{x} \in \mathbb{R}^n$ **y** $\boldsymbol{\theta} \in \mathbb{R}^n$. Son de la misma dimensión.

Dicho algoritmo expresado en pseudocódigo es el siguiente:

1. Inicializar $x^{(1)}, \dots, x^{(nm)}, \theta^{(1)}, \dots, \theta^{(nu)}$ a valores aleatorios y pequeños.
   Esto se utiliza, al igual que en las Redes Neuronales, para evitar la Rotura de la Simetría y se asegura que el algoritmo aprenda las variables "*x*" correctamente.
2. Minimizar $J(x^{(1)}, \dots, x^{(nm)}, \theta^{(1)}, \dots, \theta^{(nu)})$ utilizando el Gradiente Descendente (o algún algoritmo de optimización avanzado).
3. Dados los parámetros "*θ*" del usuario y una película con los parámetros "*x*" aprendidos, predecir el puntaje con $\theta^T x$.

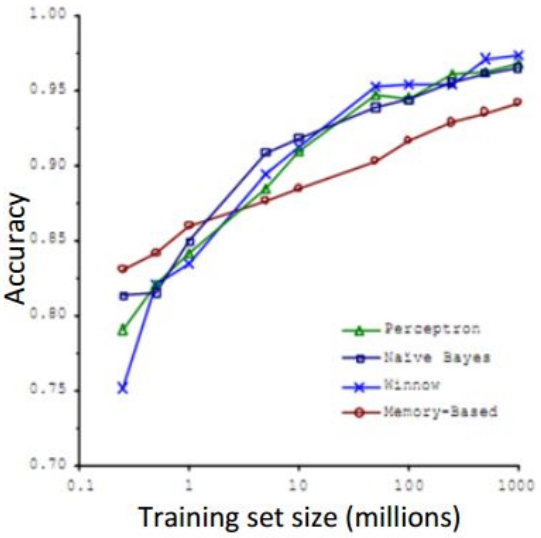
XIV. Machine Learning a gran escala

Este capítulo tratará sobre el manejo de conjuntos de datos a de gran dimensión.

La razón de esta implementación puede observarse en la gráfica de la derecha donde, a mayor cantidad de conjuntos de entrenamientos, el *rendimiento* de los algoritmos se ve beneficiado.

Esta cantidad de datos presenta un inconveniente computacional, ya que deben realizarse millones de cálculos.

Para utilizar la menor cantidad de datos posibles, puede graficarse las *Curvas de Aprendizaje*.



Gradiente Descendente Estocástico

El problema con el Gradiente Descendente común radica en que el término derivado es costoso computacionalmente si se tiene una gran cantidad de ejemplos de entrenamiento.

La diferencia del *Batch* respecto al *Estocástico* se debe principalmente a la cantidad de ejemplos de entrenamiento utilizadas, pero además posee otras diferencias:

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

El coste se calcula respecto a cada ejemplo, como el error cuadrático medio.

En cambio, la función de coste "*J*" es solo un promedio de los "*m*" costos anteriores. $$J_{train}(\theta) = \frac{1}{m} \sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

Una vez sabido esto, el algoritmo procede de la siguiente manera:

1) Se mezclan aleatoriamente las muestras.
2) Se repite un lazo for de "m" iteraciones:

$$\Theta_j := \Theta_j - \alpha(h_\Theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} (\text{for } j = 0, \ldots, n)$$

Este proceso no realiza la suma de los "m" ejemplos de entrenamiento por cada cálculo de Θ_j.

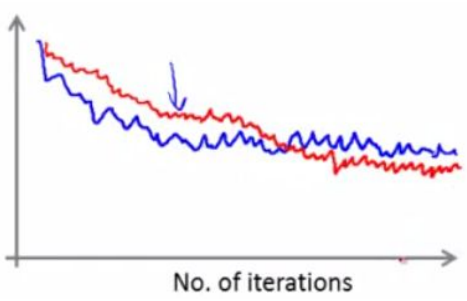
Este proceso tenderá a llegar al mínimo global de forma general, pero no siempre sucederá. Puede darse la ocasión que no se llegue a este punto, sino que varíe entre puntos cercanos a él. De todos modos puede considerarse una buena hipótesis.

Para datos donde m = 300.000.000, bastará con recorrer el lazo for entre una y diez veces para llegar a un resultado razonable.

Convergencia

La función de coste se calcula antes de actualizar θ usando solo $\{x^{(i)},y^{(i)}\}$ y se grafica el promedio de $cost(\theta,(x^{(i)},y^{(i)}))$.

Estos gráficos se ven "ruidosos", o sea que no siempre se observará un decrecimiento de la función de coste.



Un ejemplo se muestra a la izquierda, donde el rojo posee una Tasa de Aprendizaje (α) menor. Puede apreciarse una leve mejoría.

Donde se encuentra la flecha puede considerarse que la función de coste ha llegado a la periferia del mínimo local.

Utilizando un α menor, se obtendrán menores oscilaciones ya que este algoritmo converge a las cercanías del mínimo local.

Si se calculara la función de coste para una cantidad mayor de ejemplos de entrenamiento, el gráfico se vería más suavizado, no con tantas oscilaciones.

Como en las implementaciones anteriores, si el gráfico muestra una función *divergente*, será necesario disminuir α.

Gradiente Descendente Mini Batch

Este algoritmo, en varias ocasiones, puede resultar más eficiente que el procedimiento Estocástico si la implementación es vectorizada.

Say $b = 10, m = 1000$.
Repeat {
  for $i = 1, 11, 21, 31, \ldots, 991$ {
    $\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$
      (for every $j = 0, \ldots, n$)
  }
}

Por cada iteración utilizará una cantidad "*b*" (*Tamaño Mini Batch*). Este valor suele ser de 2 a 100.

Un ejemplo del algoritmo se muestra a la izquierda. El ciclo se realiza sobre todo el ejemplo "*m*" pero en saltos de a "*b*" y el gradiente sobre "*b*".

Aprendizaje en línea (Online Learning)

Se utiliza cuando es necesario ajustar datos que ingresan constantemente (como un *streaming*) a una página de internet a través de los usuarios.
El pseudo algoritmo sería de la siguiente manera:

Repetir en un lazo infinito {
    Tomar (x,y) correspondiente a cada usuario
    Actualizar θ usando solamente $(x^{(i)},y^{(i)})$:
        $\theta_j := \theta_j - \alpha\ (h_\theta(x) - y)\ x_j$ para (j = 0 , … , n)
}

MapReduce y Paralelismo de datos

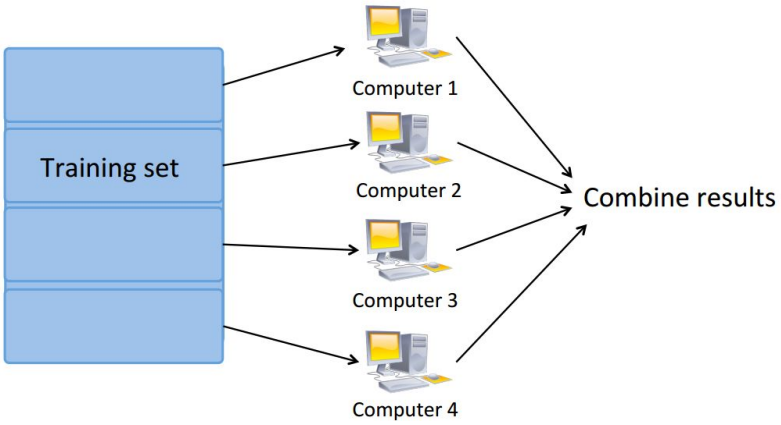
MapReduce es una téncica para el manejo paralelo de datos. Consiste en dividir una tarea extensa como la siguiente:

$$\theta_j := \theta_j - \alpha \frac{1}{400} \sum_{i=1}^{400} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

en por ejemplo, cuatro partes iguales para ser procesada cada una por una computadora.
Cada computadora calculará la sumatoria para su determinado rango del conjunto de entrenamiento y luego se los combina:

$$\Theta_j := \Theta_j - \alpha \frac{1}{z} (temp_j^{(1)} + temp_j^{(2)} + \cdots + temp_j^{(z)})$$

Gráficamente esta implementación se vería como:



Con esta implementación se logra que las computadoras reduzcan su tiempo de trabajo, obteniendo una mayor velocidad de procesamiento.
Esta técnica también puede ser utilizada en computadora con varios núcleos.
Existen aplicaciones open source que implementan esta técnica; la más conocida se llama *Apache Hadoop*.