Capítulo 1

Introducción a ciencia de datos

Programa:

No es válido usar implementaciones ya incluidas en alguna biblioteca.

Desarrollar un código que reciba un texto y devuelva el número de ocurrencias de cada caracter sin contar el espacio en blanco; por ejemplo:

```
cuenta_letras("últimamente hemos tenido varios días muy lluviosos")
```

Devuelve:

```
s     6
o     5
m     4
e     4
i     4
t     3
a     3
l     3
n     2
d     2
u     2
v     2
r     1
h     1
y     1
í     1
ú     1
```

Y otro que reciba una lista de textos y devuelva el número de ocurrencias de cada palabra:

```
txt = ['Hola, cómo estás!',
       'gana dinero, gana desde casa.',
       'Hola, puedes llamar ahora?',
       'Hola, te puedo llamar mañana?']

cuenta_palabras(txt)
```

Devuelve:

```
ahora   1
casa    1
cómo    1
desde   1
dinero  1
estás   1
gana    2
hola    3
llamar  2
mañana  1
puedes  1
puedo   1
te      1
```

También puede probarse con el texto descriptivo: https://oferta.unam.mx/ciencia-de-datos.html

---*

Introducción

Un candidato robot que quiere tu voto: https://bit.ly/2soZCiI.

¿Qué es la *inteligencia*? (http://bit.ly/75jhjs). Del latín *intellegentĭa*, es la capacidad de entender, asimilar, elaborar información y utilizarla para resolver problemas. Sin embargo, no parece existir total acuerdo para definirla (http://bit.ly/3FvhR7).

Existen diversas formas de *medir* la inteligencia como el examen de IQ (http://bit.ly/mSiVMi), aunque hay un programa que obtiene el puntaje (http://bit.ly/WpCx8o) de genios históricos (http://bit.ly/124R3DE)[1].

La Inteligencia Artificial (IA) es la inteligencia de las máquinas y la rama de la ciencia computacional que busca los medios para dotar a los sistemas informáticos de dicha inteligencia. El objetivo de la IA (fuerte) es crear una *máquina pensante* que *sea inteligente*, *tenga consciencia*, *capacidad de aprender*, *libre albedrío* y que *sea ética*. La IA es un campo joven, el término fue acuñado por John McCarthy (http://bit.ly/b9ArwT) en 1956. Ya antes Alan Turing (http://bit.ly/Hl8F) había diseñado el Test de Turing (http://bit.ly/xVoh6) como una forma de probar el comportamiento inteligente de una máquina .

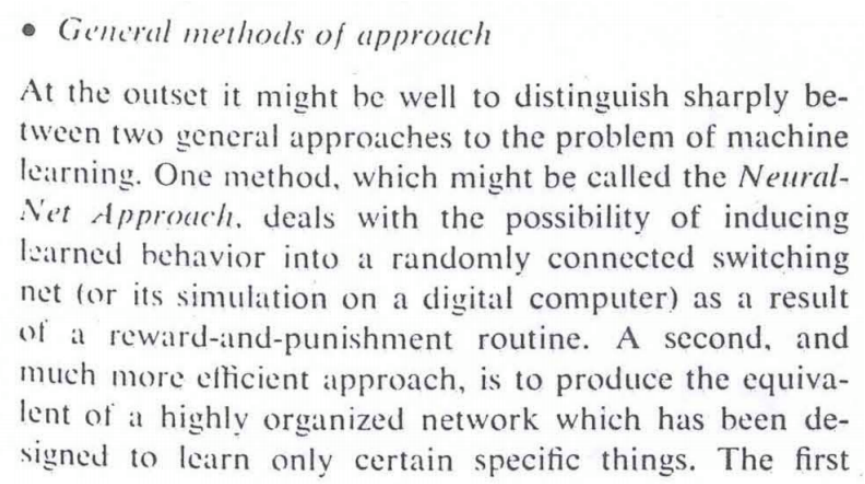
Existen profundos problemas filosóficos en IA y algunos investigadores creen que sus objetivos son imposibles o incoherentes. Esta visión es compartida entre otros por Hubert Dreyfus (http://bit.ly/OMWYce) y John Searle (http://bit.ly/fdCL2M). Aún si la IA es posible, existen consideraciones morales que deben tomarse en cuenta como la explotación de las máquinas por el hombre y si esto es o no ético. Joseph Weizenbaum (http://bit.ly/NO1er)[2] argumentó que la IA no es ética.

Recientemente Stuart Russell (http://bit.ly/1KgY9uJ), Peter Norvig, Steve Wozniak y Stephen Hawking entre otras muchas personalidades de la ciencia han firmado una *Open Letter* sobre

[1] ¿Cómo les aplicaron la prueba?

[2] Weizenbaum fue un psicólogo inventor de ELIZA (http://bit.ly/ThrCF), este programa simulaba un psicólogo en diálogo con un paciente. Inicialmente tenía una postura a favor de la IA, pero posteriormente fue un crítico de ésta.

las prioridades de investigación para una inteligencia artificial robusta y benéfica (http://futureoflife.org/AI/open_letter).



• *General methods of approach*

At the outset it might be well to distinguish sharply between two general approaches to the problem of machine learning. One method, which might be called the *Neural-Net Approach*, deals with the possibility of inducing learned behavior into a randomly connected switching net (or its simulation on a digital computer) as a result of a reward-and-punishment routine. A second, and much more efficient approach, is to produce the equivalent of a highly organized network which has been designed to learn only certain specific things. The first

Figure 1.1: ¿ML en 1959? (https://bit.ly/2ZWgjCZ)

¿Diseñar una IA que distinga entre perros y panqués?

¡Debe ser muy simple!

.

.

.

Figure 1.2: ¿Perro o panqué? (https://bit.ly/2YSneMZ)

¿Qué tal entre perros y pollo?

¡Esto sí debe ser muy simple!

.
.
.

Figure 1.3: ¿Perro o pollo? (https://bit.ly/3ya5rAC)

¿Bueno, entre perros y papas?

.
.
.

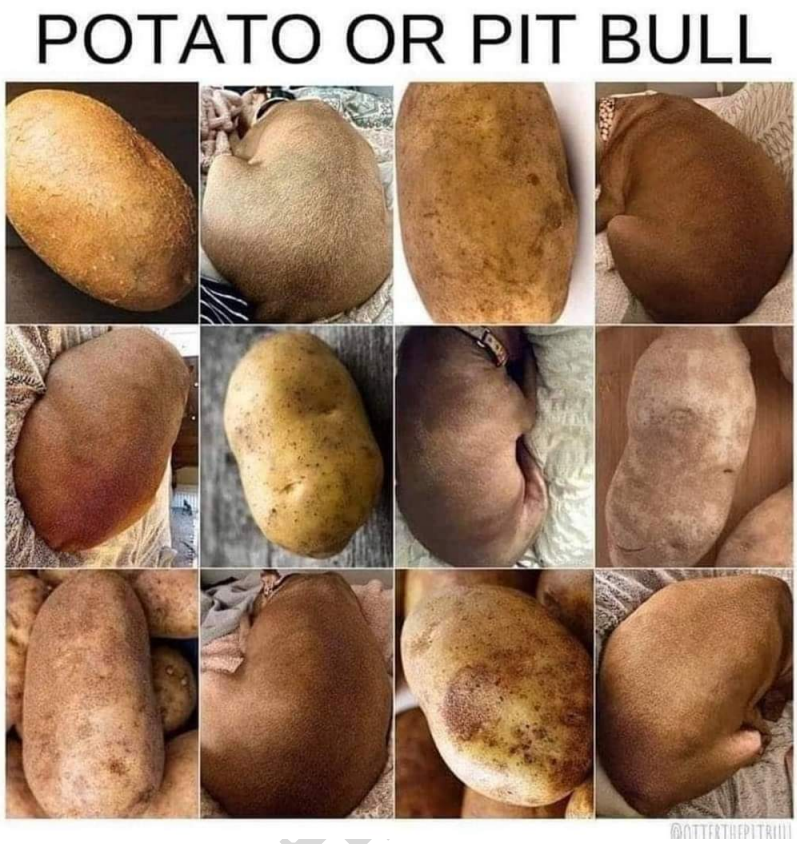


Figure 1.4: ¿Perro o papa? (https://bit.ly/3ya5rAC)

Figure 1.5: Convertir número a texto (https://bit.ly/46YtVj5)

Tarea

Leer los artículos para discutirlos en clase:

♢ "*Computing machinery and intelligence*" (http://bit.ly/2gdbpKD) de Alan Turing y elegir alguna de las *visiones opuestas* para exponer (con argumentos) si estás o no de acuerdo con ella

♢ La introducción (pags. 4-7) y el resumen ejecutivo (pags. 8-17) del Reporte de Cedric Villani "For a Meaningful Artificial Intelligence" (https://bit.ly/3P6csy4)

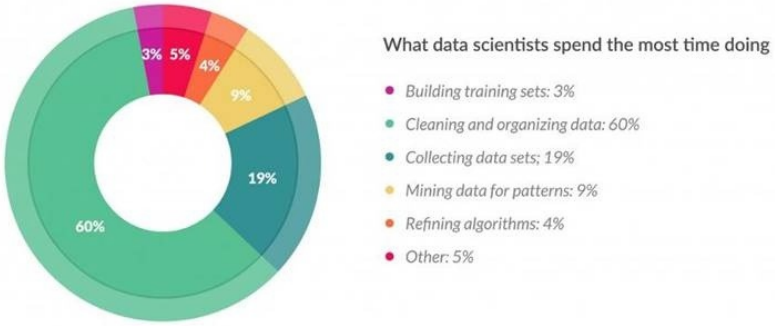
♢ Las conclusiones de cada sección (pags. 44, 76, 94, 108, 120 y 129) de la Agenda Nacional Mexicana de Inteligencia Artificial(https://www.ia2030.mx/agenda2020)

*

¿Qué es la Ciencia de Datos?

"While there is not yet a consensus on what precisely constitutes data science, three professional communities are emerging as foundational to data science: (i) Database Management to enable the transformation, conglomeration, and organization of data resources; (ii) Statistics and Machine Learning to convert data into knowledge; and (iii) Distributed and Parallel Systems to provide the computational infrastructure to carry out data analysis. Statistics and machine learning certainly play a central role in data science"[3]

¿Qué hace un Científico de Datos?



https://bit.ly/4aV4Gz2

Ejemplo: ¿Cómo obtengo $\sqrt{27}$?

What is the most mathematically involved machine learning algorithm?(https://bit.ly/3MySDwP)

Over the brief time I've spent studying the mathematics of machine learning, I've come to a realization that might seem absurd to the layman:

> *(Almost) every concept in machine learning gets mathematically more involved, the deeper you study it. Each concept is like a fractal, exhibiting ever increasing layers of complexity as you zoom into the specifics. [*]*

Figure 1.6: ML & Math

Clasificadores

Los algoritmos de clasificación se utilizan para distinguir entre objetos diferentes de un conjunto predefinido llamado universo de trabajo. Normalmente, el universo de trabajo se considera dividido en una colección K de clases $(\alpha_1, \alpha_2, \ldots \alpha_K)$, a las que pertenecen los diferentes objetos.

[3]https://magazine.amstat.org/blog/2015/10/01/asa-statement-on-the-role-of-statistics-in-data-science/

Características discriminantes

Para poder realizar el reconocimiento automático de los objetos se realiza una transformación que convierte un objeto del universo de trabajo en un vector X cuyas N componentes se llaman características discriminantes o rasgos. Estas características deben permitir discriminar a qué clases puede pertenecer cualquier objeto del universo de trabajo.

La determinación de las N características discriminantes es un proceso difícil que suele requerir del uso de la imaginación. En general, suele usarse cualquier característica que se pueda obtener de los objetos mediante algún procedimiento algorítmico.

Una vez determinadas las características discriminantes para un problema concreto, la clasificación de un objeto comienza por la obtención de su patrón. El siguiente paso consiste en determinar la proximidad o grado de pertenencia de este patrón a cada una de las clases existentes. A este efecto se definen las *funciones discriminantes* o *funciones de decisión* como aquellas funciones que asignan a un objeto un grado de semejanza respecto a cada una de las diferentes clases.

Criterios para la selección de características

En general se busca el conjunto mínimo de características que permiten determinar de manera unívoca a qué clase pertenecen todos los objetos del universo de trabajo. Una mala elección de las características discriminantes puede hacer que el sistema sea innecesariamente caro y lento, o que sea imposible construir un clasificador para resolver un problema utilizando tales características.

Usualmente se exigen cinco propiedades que deben poseer las características que se seleccionen: *economía*, *velocidad*, *fiabilidad*, *capacidad discriminante* e *independencia* con respecto a otras características.

Economía El mecanismo preciso para el cálculo o la obtención de las características discriminantes debe tener un costo *razonable*.

Velocidad El tiempo de cálculo no debe superar un *umbral* que lo haga inviable.

Independencia Las características no deben estar correlacionadas entre ellas. Una característica que depende fuertemente del resto no añade información discriminante y por tanto puede eliminarse sin que esto suponga ninguna pérdida de capacidad discriminante.

Fiabilidad La fiabilidad implica que objetos de la misma clase deben tener vectores de características con valores numéricos similares. Esto se cumple si los vectores de características de una clase tienen poca dispersión.

Capacidad discriminante La capacidad discriminante de una característica determinada se puede describir como una propiedad que asegura que patrones de clases distintas tienen valores numéricos claramente diferenciados.

Tipos de algoritmos de clasificación

Los algoritmos usados se pueden clasificar de acuerdo a diferentes criterios.

Clasificadores *a priori* y *a posteriori*

Los *clasificadores a priori* construyen el clasificador en un solo paso, utilizando una muestra de aprendizaje para el cálculo de las funciones discriminantes y un cálculo exacto.

Los *clasificadores a posteriori* o con *aprendizaje* se construyen siguiendo un procedimiento iterativo de entrenamiento, en el cual el clasificador aprende a reconocer de una manera progresiva los patrones de las muestras de aprendizaje. Para ello suelen utilizar técnicas aproximadas, lo que implica que el tiempo de aprendizaje puede no ser despreciable.

Clasificadores supervisados y no supervisados

Tomando en cuenta la información que se proporciona en el proceso de construcción del clasificador se puede hablar de dos tipos de clasificadores: *supervisados* o *no supervisados*.

En los **supervisados**, el supervisor lleva a cabo las etapas en la construcción del clasificador: determinación de las clases, elección y prueba de las características discriminantes, selección de la muestra, cálculo de funciones discriminantes y prueba del clasificador.

En los **no supervisados** este proceso se realiza de manera automática, sin la necesidad de ningún supervisor externo. Para ello se emplean técnicas de agrupamiento, gracias a las cuales el sistema selecciona y aprende los patrones que poseen características similares, determinándose automáticamente las clases.

Clasificadores deterministas y no deterministas

De acuerdo a la forma en que se distribuyen los patrones de la muestra se puede hablar de que se cumple o no la *hipótesis determinista*: cada clase se puede representar por un único vector que se llama *prototipo representante* de la clase. Según esta hipótesis se puede hablar de dos tipos de clasificadores: *clasificadores deterministas* y *clasificadores no deterministas*.

Dependiendo de las características seleccionadas puede ser necesario el uso de uno u otro tipo. Cuando las características elegidas hacen que los patrones de clases diferentes se sitúen en regiones disjuntas, los clasificadores deterministas darán buenos resultados. Si las regiones no son disjuntas ofrecerán mejores resultados los clasificadores no deterministas.

Clasificador de distancia euclideana determinista *a priori*

Es un clasificador *determinístico*, *supervisado* y *a priori*. Se basa en el cálculo de un prototipo o *centroide* para cada una de las K clases en las que se divide el universo de trabajo. Este prototipo puede verse como un representante "ejemplar"de cómo debería de ser un vector de características de esa clase. Así, ante un patrón desconocido se calcula la distancia euclídea del patrón que se desea clasificar a cada uno de los K prototipos. Un patrón desconocido X se clasificará como correspondiente a la clase cuyo prototipo esté a menor distancia según la distancia euclideana:

$$d_E = \sqrt{X^T \cdot X - 2 \cdot X^T \cdot Z_K + Z_K^T \cdot Z_K}$$

Así, el clasificador euclideano divide el espacio de características en regiones mediante *hiperplanos equidistantes* de los centroides. La figura 1.7 presenta el caso de 3 clases y 2 características. Cada línea punteada separa los puntos del espacio más cercano a cada uno de los centroides (representados por puntos negros)

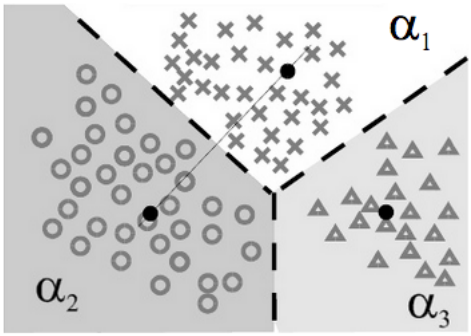


Figure 1.7: Ejemplo de separación entre clases

Para el caso de K clases: $\alpha_1, \alpha_2, \ldots \alpha_K$, se necesitan K prototipos: $Z_1, Z_2, \ldots Z_K$. Al clasificar un patrón se sigue el esquema de la figura 1.8. Según este esquema, para clasificar el patrón X se calcula la distancia del vector de características X a los vectores de características de cada uno de los K prototipos, clasificando X como perteneciente a la clase cuyo prototipo esté más próximo.

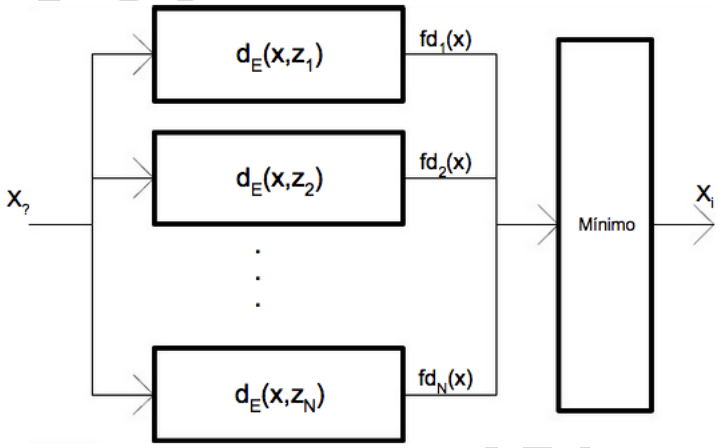


Figure 1.8: Proceso de clasificación

La función discriminante se puede simplificar eliminando la raíz cuadrada, ya que es una transformación que mantiene la relación de distancia. También se puede eliminar el término $X^T \cdot X$ pues es igual para todas las clases. Finalmente suele cambiarse el signo y dividirse entre 2 obteniendo la expresión.

$$fd_K(X) = X^T \cdot Z_K - \tfrac{1}{2} Z_K^T \cdot Z_K \qquad \forall k \in K$$

Debido al cambio de signo será ahora la función discriminante con valor máximo la que marcará la clase a la que pertenece el patrón X.

El proceso de cálculo de los prototipos es un proceso *heurístico*: suele usarse la media ponderada de un conjunto de P elementos de esa clase. Esto hace que, para el cálculo de prototipos, se necesite un conjunto de muestra que incluya varios individuos para cada una de las N clases. El tamaño de este conjunto debe ser tal que sea representativo de la clase de elementos a la que corresponda.

Ejemplo:

Supóngase que se tienen que distinguir patrones de dos clases, α_1 y α_2, y que se dispone de la siguiente muestra:

$$\omega_1 = \left\{ \begin{pmatrix} 1 \\ 5 \\ 6 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \\ 4 \\ -2 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 5 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \\ 5 \\ 2 \end{pmatrix} \right\} \text{ y } \omega_2 = \left\{ \begin{pmatrix} 6 \\ 8 \\ -1 \\ 6 \end{pmatrix}, \begin{pmatrix} 7 \\ 9 \\ 3 \\ 6 \end{pmatrix}, \begin{pmatrix} 8 \\ 7 \\ 1 \\ 9 \end{pmatrix} \right\}$$

El primer paso consiste en calcular los centroides de las dos clases:

$$z_1 = \left\{ \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix} \right\} \text{ y } z_2 = \left\{ \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix} \right\}$$

Posteriormente se calculan las funciones discriminantes:

$$fd_1 = X^T \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix} - \tfrac{1}{2} \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix} \text{ y } fd_2 = X^T \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix} - \tfrac{1}{2} \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix}^T \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix}$$

Con estas funciones se pueden clasificar objetos nuevos, por ejemplo, para el vector $X = (3, 1, 3, 1)$

$$fd_1 = \begin{pmatrix} 3 \\ 1 \\ 3 \\ 1 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix} - \tfrac{1}{2} \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix}^T \cdot \begin{pmatrix} 1 \\ 3 \\ 5 \\ 1 \end{pmatrix} = 3 + 3 + 15 + 1 - \tfrac{1}{2}(1 + 9 + 25 + 1) = 4$$

$$fd_2 = \begin{pmatrix} 3 \\ 1 \\ 3 \\ 1 \end{pmatrix}^T \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix} - \tfrac{1}{2} \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix}^T \cdot \begin{pmatrix} 7 \\ 8 \\ 1 \\ 7 \end{pmatrix} = 21 + 8 + 3 + 7 - \tfrac{1}{2}(49 + 64 + 1 + 49) = -42.5$$

Finalmente: $fd_1 > fd_2 \Rightarrow X \in \alpha_1$

Ejercicio, tarea:

Obtener las funciones discriminantes para las muestras:
$\omega_1 = \{(0.5, 10.5), (1, 12.5), (3, 10.5), (3, 12.5), (3, 14.5), (3, 18), (5, 18), (5, 16), (5, 14.5), (5, 13)\}$
$\omega_2 = \{(6, 9), (8, 10), (9, 11), (8.5, 12), (7, 13.5)\, (8, 16)\}$
Representarlo gráficamente.

*

```python
# Clasificador euclideano determinista y a priori
import numpy as np
from sympy import Matrix, symbols, simplify
def clasif_e(samples):
  n_samples = len(samples)
  # Vector genérico con Sympy
  X = Matrix( [symbols('x'+str(i+1)) for i in range(samples[0].shape[0])] )
  print(' Variables : ',X)
  fds = []
  for s in samples:
    m = Matrix(np.mean(s, axis=1))
    fds.append( simplify(X.T*m - (m.T*m)/2) )
  return fds
```

Clasificador estadístico *a priori*

En las situaciones en que los vectores de alguna clase presenten una dispersión significativa respecto a la media, o en aquéllas en las que no existe posible separación lineal entre las clases, puede ofrecer mejores resultados la sustitución de la distancia euclideana por la *distancia de Mahalanobis* (http://bit.ly/ciIlZ0).

Esta medida toma en cuenta la desviación estándar de los vectores de características de los patrones de la muestra y puede proporcionar regiones de separación entre clases que sigan curvas cónicas. Por ello, este clasificador ofrece más garantías al tratar de separar patrones para los que no se encuentra una separación lineal. Además el clasificador estadístico proporciona *probabilidad de pertenencia* a las clases, por lo que se debe incluir en el grupo de los clasificadores no deterministas.

La distancia de Mahalanobis de un objeto X a una clase α_k se determina por la expresión:

$$d_M\left(X, \alpha_k\right) = \left(X - m_k\right)^T \cdot C_k^{-1} \cdot \left(X - m_k\right)$$

Donde m_k es la media y C_k es la matriz de covarianzas (http://bit.ly/18wTUZu) de la clase k. Más adelante se explica cómo aparece esta distancia de manera natural al enfocar el problema de la clasificación desde un punto de vista probabilista (figura 1.9).

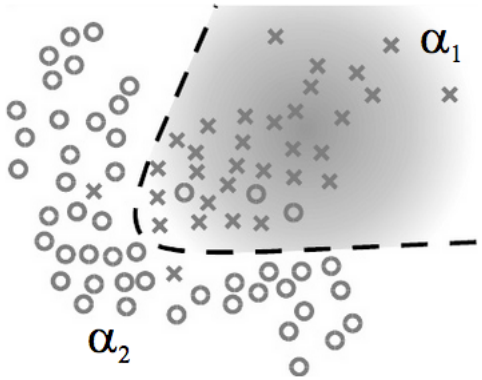


Figure 1.9: En un clasificador estadístico la región de separación no es determinista. Además crea separaciones no lineales de dos clases (curvas cónicas como las parábolas, elipses, hipérbolas y, por su puesto, rectas).

Clasificación probabilista

Cuando existe diferente dispersión de los valores de una característica en dos o más clases, una medida de la distancia que tenga en cuenta la desviación estándar de la clase ofrecerá mejores resultados que otra que sólo tenga en cuenta la distancia euclideana entre los centroides de las clases.

La figura 1.10 ejemplifica la disposición de los patrones de dos clases respecto de una característica particular. En la figura se aprecia que la función discriminante no equidista de los centroides de α_1 y α_2 sino que está más cerca del centroide de α_2 porque su desviación estándar es menor.

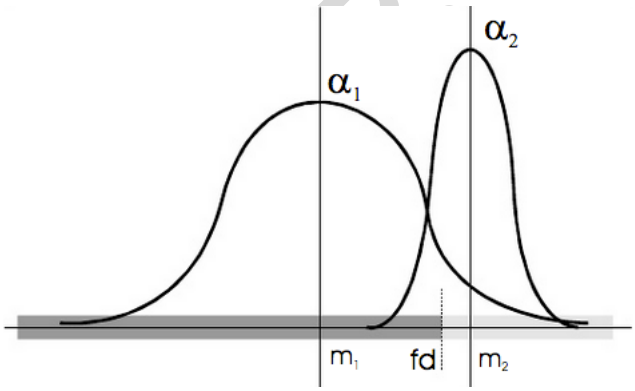


Figure 1.10: La función discriminante depende de la desviación estándar de las clases.

El teorema de Bayes es la base para obtener la función discriminante:

$$P\left(\alpha_i|X\right) = \frac{P(X|\alpha_i)P(\alpha_i)}{P(X)}$$

Donde:

♢ $P\left(\alpha_i\right)$ es la probabilidad de que un patrón al azar pertenezca a la clase α_i.

♢ $P(X)$ es la probabilidad de que se presente exactamente un patrón con el vector de características X. Se cumple que $P(X) = \sum_{k=1}^{K} P(X|\alpha_k)\, P(\alpha_k)$.

♢ $P(X|\alpha_i)$ es la *probabilidad a priori*: probabilidad de que sabiendo que el objeto X a clasificar pertenece a la clase α_i, sea asignado a esa clase.

♢ $P(\alpha_i|X)$ se conoce como *probabilidad a posteriori* y corresponde a la probabilidad de que un X conocido pertenezca a la clase α_i.

Utilizando la probabilidad *a posteriori* se puede construir un clasificador no determinista que asigne a un patrón determinado la probabilidad de pertenencia a cada clase.

Además, aunque el clasificador estadístico sea no determinista, cuando se precise una respuesta concreta puede transformarse en determinista añadiendo la siguiente regla:

$$X \in \alpha_i \Leftrightarrow P(X|\alpha_i) > P(X|\alpha_j) \qquad \forall i \neq j, j = 1, 2, \ldots N$$

Como $P(X)$ es un término constante para todos los cálculos de $P(\alpha_i|X)$, puede simplificarse eliminando ese factor. Con esto la función discriminante para la clase α_i es:

$$fd_i(X) = P(X|\alpha_i)\, P(\alpha_i) \qquad \forall i = 1, 2, \ldots N$$

Y:

$$X \in \alpha_i \Leftrightarrow fd_i(X) > fd_i(X) \qquad \forall i \neq j, j = 1, 2, \ldots N$$

Diseño de un clasificador estadístico *a priori*

El diseño de un clasificador exige conocer la distribución de probabilidad $P(X|\alpha_i)$. Para el caso más simple la distribución de probabilidad $P(X|\alpha_i)$ tiene distribución normal o gaussiana unidimensional. Es por esto que suele hacerse la hipótesis de normalidad. En caso de no cumplirse debe proponerse una distribución de probabilidad adecuada. El siguiente desarrollo supone la hipótesis de normalidad.

La función de probabilidad para un vector con una sola característica con distribución normal se determina con:

$$P(X|\alpha_i) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{-\frac{(X-m_i)^2}{2\sigma_i}} \qquad \forall i = 1, 2, \ldots N$$

Donde m_i es la media y σ_i la desviación estándar de la distribución de la clase α_i.

Si la probabilidad de obtener objetos de diferentes clases es la misma los cálculos se simplifican: si $P(\alpha_i) = P(\alpha_k)$ para cualquier clase i y k este término no influye en el cálculo. Así, para el caso de dos clases equiprobables se obtiene:

$$X \in \alpha_1 \Leftrightarrow P(\alpha_1|X) > P(\alpha_2|X) \Leftrightarrow P(X|\alpha_1) > P(X|\alpha_2)$$

Esta expresión puede simplificarse sustituyendo $P(X|\alpha_i)$:

$$X \in \alpha_1 \Leftrightarrow \frac{1}{\sqrt{2\pi}\sigma_1} e^{-\frac{(X-m_1)^2}{2\sigma_1}} > \frac{1}{\sqrt{2\pi}\sigma_2} e^{-\frac{(X-m_2)^2}{2\sigma_2}}$$

Tomando logaritmos y normalizando:

$$X \in \alpha_1 \Leftrightarrow \frac{(X-m_1)^2}{2\sigma_1} - \ln(\sigma_1) < \frac{(X-m_2)^2}{2\sigma_2} - \ln(\sigma_2)$$

Para el caso n-dimensional la función de probabilidad normal es

$$P(X|\alpha_k) = \frac{1}{(2\pi)^{\frac{n}{2}} \cdot |C_k|^{\frac{1}{2}}} e^{-\frac{1}{2}\left(X-\vec{m}_k\right)^T \cdot C_k^{-1} \cdot \left(X-\vec{m}_k\right)} \qquad \forall k = 1, 2, \ldots N$$

Donde C_k es la matriz de covarianzas

Teniendo en cuenta los resultados precedentes, la función general discriminante para una clase, cuando las probabilidades de ocurrencia de las diferentes clases son iguales, y tras hacer logaritmos para eliminar el termino exponencial, resulta:

$$fd_i(X) = -\frac{1}{2} X^T \cdot C_i^{-1} \cdot X + X^T \cdot C_i^{-1} \cdot \vec{m}_i - \frac{1}{2} \vec{m}_i^T \cdot C_i^{-1} \cdot \vec{m}_i - \frac{1}{2} \ln |C_i| \qquad \forall i = 1, 2, \ldots N$$

Si las matrices de covarianza son iguales la función discriminante se simplifica a:

$$fd_i(X) = X^T \cdot C_i^{-1} \cdot \vec{m}_i - \frac{1}{2} \vec{m}_i^T \cdot C_i^{-1} \cdot \vec{m}_i \qquad \forall i = 1, 2, \ldots N$$

Si además la matriz de covarianza tiene valores cercanos a cero, y con todas las desviaciones estándar iguales se obtiene una formula idéntica a la del clasificador euclideano:

$$fd_i(X) = X^T \cdot \vec{m}_i - \frac{1}{2} \vec{m}_i^T \cdot \vec{m}_i \qquad \forall i = 1, 2, \ldots N$$

Hay que señalar que en la práctica los valores que se obtienen para las desviaciones nunca son exactamente iguales, ni las covarianzas son exactamente cero, pero estas reglas se aplican igualmente si se aproximan *suficientemente* a tales valores.

Ejemplo:

Se dispone de la siguiente muestra, correspondiente a objetos de dos clases α_1 y α_2 equiprobables. Se desea construir un clasificador de bayesiano que asigne correctamente los patrones de la muestra sabiendo que los patrones tienen características que siguen distribuciones normales:

$$\omega_1 = \left\{ \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \begin{pmatrix} 2 \\ 2 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ 3 \end{pmatrix}, \begin{pmatrix} 3 \\ 2 \end{pmatrix} \right\}$$

$$\omega_2 = \left\{ \begin{pmatrix} 8 \\ 10 \end{pmatrix}, \begin{pmatrix} 9 \\ 8 \end{pmatrix}, \begin{pmatrix} 9 \\ 9 \end{pmatrix}, \begin{pmatrix} 8 \\ 9 \end{pmatrix}, \begin{pmatrix} 7 \\ 9 \end{pmatrix} \right\}$$

Se obtienen las medias:

$$\vec{m}_1 = \begin{pmatrix} 2.2 \\ 2 \end{pmatrix} \text{ y } \vec{m}_2 = \begin{pmatrix} 8.2 \\ 9 \end{pmatrix}$$

Y matrices de covarianzas:

$$C_1 = \begin{pmatrix} 0.56 & -0.2 \\ -0.2 & 0.4 \end{pmatrix} \text{ y } C_2 = \begin{pmatrix} 0.56 & -0.2 \\ -0.2 & 0.4 \end{pmatrix}$$

Como se observa que $C_1 = C_2$ y además se ha dicho en el enunciado que los patrones de ambas clases son equiprobables y sus distribuciones normales se puede usar la función discriminante:

$$fd_i(X) = X^T \cdot C_i^{-1} \cdot \vec{m}_i - \frac{1}{2} \vec{m}_i^T \cdot C_i^{-1} \cdot \vec{m}_i$$

con:

$$C^{-1} = \begin{pmatrix} \frac{50}{23} & \frac{25}{23} \\ \frac{25}{23} & \frac{70}{23} \end{pmatrix}$$

Por tanto:

$$fd_1(X) = \frac{160}{23}x_1 + \frac{195}{23}x_2 - \frac{378}{23}$$

$$fd_2(X) = \frac{635}{23}x_1 + \frac{835}{23}x_2 - \frac{6361}{23}$$

Tarea:

Obtener las funciones discriminantes para las muestras:
$\omega_1 = \{(0.5, 10.5), (1, 12.5), (3, 10.5), (3, 12.5), (3, 14.5), (3, 18), (5, 18), (5, 16), (5, 14.5), (5, 13)\}$
$\omega_2 = \{(6, 9), (8, 10), (9, 11), (8.5, 12), (7, 13.5)\, (8, 16)\}$
Nota: Las matrices de covarianzas no son iguales: se obtendrán funciones discriminantes cónicas.

*

```python
# Clasificador de distacia de Mahalanobis
import numpy as np
from sympy import Matrix, symbols, simplify, log
def clasif_m(samples):
  nsamples = len(samples)
  X = Matrix( [symbols('x'+str(i+1)) for i in range(samples[0].shape[0])] )
  print('Variables : ',X)
  fds = []
  for s in samples:
    m = Matrix(np.mean(s, axis=1))
    m_cov = np.cov(s, bias=True)
    print(m_cov)
    m_ci = Matrix(m_cov).inv()
    fds.append( simplify((X.T*m_ci*X)/-2 + X.T*m_ci*m - (m.T*m_ci*m)/2 - Matrix([log(m_ci.det())/2]) ) )
  return fds
```

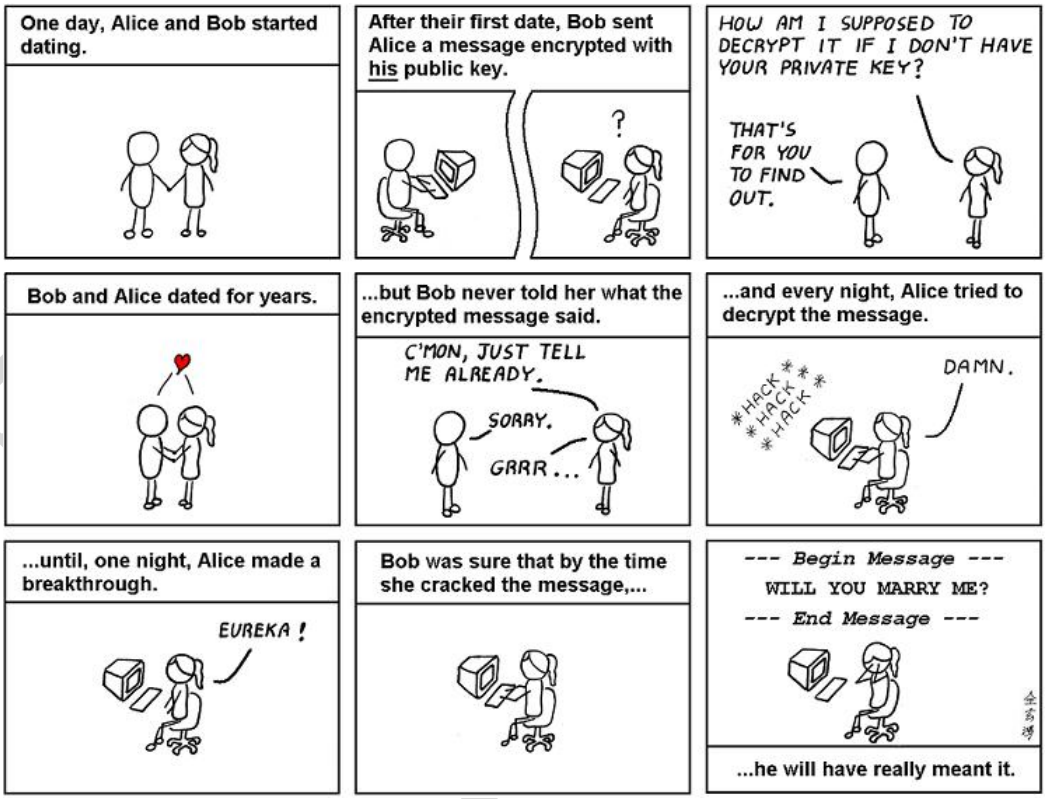


Figura 1.11: Amor criptográfico: Alice & Bob

Programa:

Consiste en hacer un código que se encargue de cifrar y descifrar texto de acuerdo al método mostrado a continuación:

1. Este tipo de cifrado por columna con palabra clave consiste en formar una tabla con tantas columnas como letras tenga la palabra clave; a continuación, se escribe el texto en la tabla de izquierda a derecha y de arriba hacia abajo (sin espacios y, si hace falta, se rellenan los espacios de la última fila con algún caracter):

```
Texto: LA CRIPTOGRAFIA ES ROMANTICA
Clave: HOLA
```

H	O	L	A
L	A	C	R
I	P	T	O
G	R	A	F
I	A	E	S
R	O	M	A
N	T	I	C
A	S	S	S

2. A continuación se reordenan las columnas alfabéticamente de acuerdo a la palabra clave (si hay repetición, el criterio de desempate es el orden de aparición en la palabra):

A	H	L	O
R	L	C	A
O	I	T	P
F	G	A	R
S	I	E	A
A	R	M	O
C	N	I	T
S	A	S	S

3. Finalmente se toman los caracteres por columna de arriba hacia abajo y de izquierda a derecha obteniendo finalmente el texto codificado:

```
ROFSACSLIGIRNACTAEMISAPRAOTS
```

Para descifrar un texto codificado con este método, es necesario saber la palabra clave, y a continuación se aplican las operaciones siguientes, mostradas para descrifrar el ejemplo anterior:

```
Texto: ROFSACSLIGIRNACTAEMISAPRAOTS
Clave: HOLA
```

1. Se divide el texto en tantas partes como letras tiene la palabra clave (es exacta):

```
Grupos:    ROFSACS    LIGIRNA    CTAEMIS    APRAOTS
```

2. Se ordena alfabéticamente la palabra clave:

```
   Clave:  HOLA
Ordenada:  AHLO
```

3. Se coloca cada grupo bajo cada letra de la palabra clave ordenada:

A	H	L	O
R	L	C	A
O	I	T	P
F	G	A	R
S	I	E	A
A	R	M	O
C	N	I	T
S	A	S	S

4. Se reacomoda la palabra clave junto con su columna correspondiente:

H	O	L	A
L	A	C	R
I	P	T	O
G	R	A	F
I	A	E	S
R	O	M	A
N	T	I	C
A	S	S	S

5. Se concatena cada línea de la tabla para obtener el texto el claro:

```
LACRIPTOGRAFIAESROMANTICASSS
```

---

*

Neuronas artificiales y el modelo de McCulloch-Pitts

La idea origial del *perceptrón* se remonta al trabajo de Warren McCulloch (http://bit.ly/1BVMCxa) y Walter Pitts (http://bit.ly/1G7QKkd) en 1943, quienes observaron una analogía entre las neuronas biológicas y compuertas lógicas con salida binaria. Intuitivamente, una neurona puede verse como una subunidad de una red neuronal en un cerebro biológico. Las señales de magnitud variable entran por las dendritas; estas señales se acumulan en el cuerpo de la célula y, si el acumulado sobrepasa el umbral establecido, se genera una señal de salida que se entrega al axón.

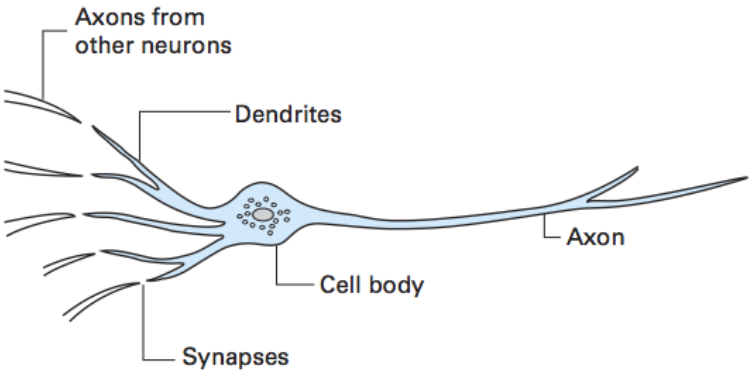


Figure 1.12: Esquema de una neurona biológica

Perceptrón de Frank Rosenblatt

Posteriormente, en 1957 Frank Rosenblatt (http://bit.ly/1CBgm5d) publicó el primer algoritmo para un *perceptrón de aprendizaje*. La idea básica es definir un algoritmo que aprende los valores de los pesos w que serán multiplicados con las características de entrada para poder decidir si la neurona *dispara* o no. El perceptrón es un clasificador de aprendizaje *supervisado*, *determinista* y *a posteriori*.

Función escalón Antes de describir a detalle el algoritmo de aprendizaje, definiremos algunos conceptos auxiliares. Primero, llamaremos a las clases positiva y negativa para nuestra clasificación binaria como 1 y -1 respectivamente. A continuación, definimos una función de activación $g(z)$ que toma una combinación lineal de las entradas x y los pesos w como entrada $z = w_1x_1 + \cdots + w_nx_n$ y, si $g(z)$ es mayor que el umbral definido θ, se obtiene 1 y -1 en otro caso. Esta función de activación se conoce como "función escalón unitario" o función escalón de Heaviside.

$$g(z) = \begin{cases} 1 & \text{si } z \geq \theta \\ -1 & \text{en otro caso} \end{cases} \text{ y } z = w_1x_1 + \cdots + w_nx_n = \sum_{i=1}^{n} w_ix_i$$

Además, se suele definir $w_0 = \theta$ y $x_0 = 1$. De este modo:

$$g(z) = \begin{cases} 1 & \text{si } z \geq \theta \\ -1 & \text{en otro caso} \end{cases} \text{ y } z = w_0x_0 + w_1x_1 + \cdots + w_nx_n = \sum_{i=0}^{n} w_ix_i$$

La regla del perceptrón de aprendizaje La idea tras del preceptrón de *umbral* es simular el funcionamiento de una célula en el cerebro: *dispara* o no. En resumen: un perceptrón recibe múltiples señales de entrada y, si la suma de las señales de entrada (multiplicadas por el peso respectivo) sobrepasa cierto umbral, entrega una señal, si no pasa el umbral, queda en *silencio*.

Este es el primer algoritmo de *aprendizaje de máquina*, dada la idea de Frank Rosenblatt, conocida como *regla de aprendizaje*: el perceptrón aprenderá los pesos para cada señal de entrada para poder dibujar un límite de decisión que nos permita discriminar entre dos clases linealmente separables.

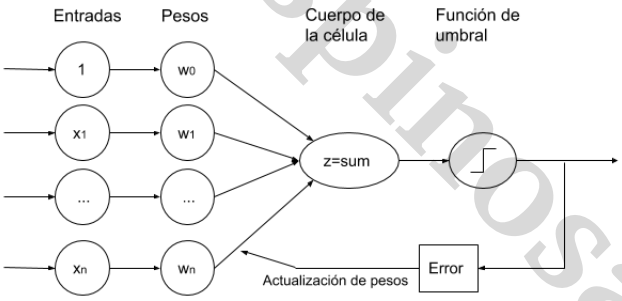


Figure 1.13: Esquema del perceptrón de Rosenblatt

La regla del perceptrón de Rosenblatt es bastante simple y puede resumirse en los pasos del algoritmo 1.1.

Algorithm 1.1 Regla del perceptrón de Rosenblatt

inicializar los pesos a 0 o un número aleatorio *pequeño*
para cada muestra de entrenamiento $x^{(i)}$:
    calcular el valor de salida $\hat{y}^{(i)}$
    actualizar pesos

El valor de salida es el predicho por la función escalón definida previamente y la actualización del peso se obtiene como $w_j = w_j + \Delta w_j x_j^{(i)}$. El valor para actualizar los pesos en cada incremento se obtiene mediante la regla de aprendizaje:

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right)$$

Donde η es la tasa (razón) de aprendizaje (una constante entre 0.0 y 1.0); $y^{(i)}$ es la clase a la que pertenece la muestra y $\hat{y}^{(i)}$ es la salida que predice el perceptrón en el paso actual. Es importante notar que el vector de pesos se actualiza *simultáneamente*.

En particular, para un conjunto de datos de 2 dimensiones, la actualización se obtiene como:

$$\begin{aligned} \Delta w_0 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) \\ \Delta w_1 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_1^{(i)} \\ \Delta w_2 &= \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_2^{(i)} \end{aligned}$$

Ejemplo, tarea

Utilizando el valor $\eta = 0.1$, aplicar el algoritmo de aprendizaje del perceptrón para una neurona artificial que calcule la función booleana NAND con 2 parámetros definida como:

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	−1

```
tropimac:nn lalo$ python nnej0.py
pesos [-0.2 -0.2 -0.2]
pesos [ 0.  -0.4 -0.2]
pesos [ 0.2 -0.4 -0.2]
pesos [ 0.2 -0.4 -0.4]
pesos [ 0.4 -0.4 -0.2]
pesos [ 0.4 -0.4 -0.2]
Pesos: [ 0.4 -0.4 -0.2]
```

Figure 1.14: Actualizaciones de los pesos en el ejemplo

*

Implementación *desde cero* del Perceptrón

Realizaremos una implementación orientada a objetos de este algoritmo

```python
import numpy as np
class Perceptron():
  def __init__(self, eta=0.1, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

  def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    if self.random_state is None:
      self.w_ = np.zeros(1+len(X[1]))
    else:
      self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1+X.shape[1])
    self.errors_ = []
    for _ in range(self.n_iter):
      errors = 0
      for xi,yi in zip(X,y):
        update = self.eta * (yi - self.predict(xi))
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
      self.errors_.append(errors)
      print('pesos : ',self.w_)
    return self

  def net_input(self, X):
    return np.dot(X, self.w_[1:]) + self.w_[0]

  def predict(self,X):
    return np.where(self.net_input(X) >= 0.0, 1, -1)
```

Figure 1.15: Código del Perceptrón

Es recomendable no iniciar los pesos a cero dado que la tasa de aprendizaje (η) sólo tiene un efecto en la clasificación si los pesos se inicializan a valores diferentes a cero: si todos los pesos son cero inicialmente, el parámetro η afectará solamente la *escala* del vector de pesos, pero **no** su *dirección*.

Probando la implementación con la NAND de 2 parámetros:

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([1,1,1,-1])

ppn = Perceptron(n_iter=6, eta=0.1)
ppn.fit(X, y)
print('Pesos: %s' % ppn.w_)
```

```
Pesos: [ 0.41624345 -0.40611756 -0.20528172]
```

Con los pesos inicializados a 0:

```
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([1,1,1,-1])

ppn = Perceptron(n_iter=6, eta=0.1, random_state=None)
ppn.fit(X, y)
print('Pesos: %s' % ppn.w_)
```

```
Pesos: [ 0.4 -0.4 -0.2]
```

Problemas con el perceptrón

El principal problema del perceptrón, el mismo Rosenblatt probó matemáticamente es que la regla de aprendizaje converge si las dos clases pueden separarse linealmente, pero no se puede garantizar su convergencia si no se cumple esta condición.

Neural network zoo: http://www.asimovinstitute.org/neural-network-zoo

Ahora probaremos la implementación con un conjunto más interesante: Iris[4]. Por el momento sólo se utilizarán dos clases: Virgínica y Versicolor; la regla del preceptrón no se restringe a dos dimensiones, pero para simplificar la visualización sólo se usarán las características *sepal length* y *petal length*. De igual forma, sólo se consideran dos tipos de flor para simplificar el ejemplo; sin embargo, la regla del perceptrón puede extenderse para ser usada en problemas multi-clase con la técnica *One-versus-All* (OvA); también conocida como *One-versus-Rest* (OvR): se toma cada clase para entrenar un perceptrón, considerando dicha clase como positiva y el resto como negativas; de esta forma se obtienen n clasificadores binarios y para clasificar una muestra nueva, se utilizan los n clasificadores para asignar la etiqueta de la clase con mayor confianza para dicha muestra.

[4]Presentado inicialmente por Ronald A. Fischer (https://bit.ly/3f9e6KF) en 1936 con el algoritmo LDA que veremos posteriormente.

```
import pandas as pd
# https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
df = pd.read_csv('https://bit.ly/38XWXS4', header=None)
df.tail()
```

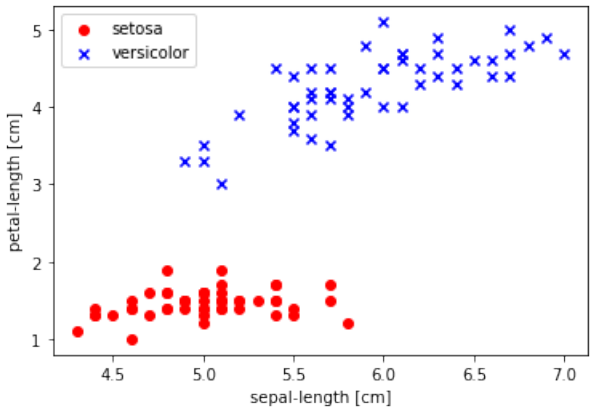
```
        0       1       2       3       4
145     6.7     3.0     5.2     2.3     Iris-virginica
146     6.3     2.5     5.0     1.9     Iris-virginica
147     6.5     3.0     5.2     2.0     Iris-virginica
148     6.2     3.4     5.4     2.3     Iris-virginica
149     5.9     3.0     5.1     1.8     Iris-virginica
```

Posteriormente extraemos las 100 primeras muestras correspondientes a 50 de *Iris-setosa* y 50 de *Iris-versicolor*, además de convertir las etiquetas de clase en dos números enteros: $1 \Rightarrow$ *versicolor* y $-1 \Rightarrow$ *setosa* asignándolas al vector y:

```
import numpy as np

X = df.iloc[0:100, [0,2]].values
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
```

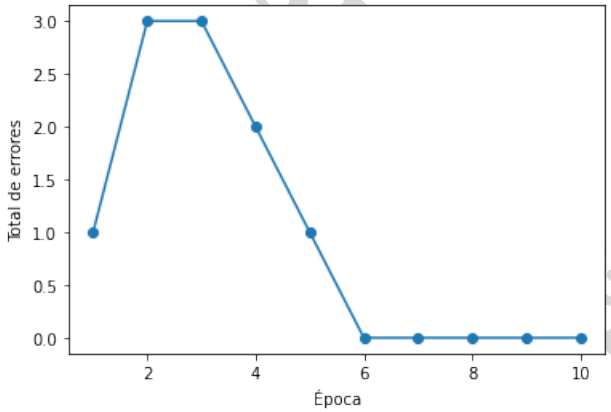
Graficando las muestras:

```
import matplotlib.pyplot as plt
plt.scatter(X[:50,0],X[:50,1],color='red',marker='o',label='setosa')
plt.scatter(X[50:100,0],X[50:100,1],color='blue',marker='x',label='versicolor'
plt.xlabel('sepal-length [cm]')
plt.ylabel('petal-length [cm]')
plt.legend(loc='upper left')
plt.show()
```



Ahora entrenamos al perceptrón con este subconjunto de datos de Iris; además graficaremos las clasificaciones erróneas en cada *época* (*epoch*) para verificar si el algoritmo ha encontrado una frontera de decisión que separe las dos clase:

```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1,len(ppn.errors_)+1), ppn.errors_, marker='o')
plt.xlabel('Época')
plt.ylabel('Total de errores')
plt.show()
```



Se observa que después de 6 épocas el perceptrón ha convergido y puede separar ambas clases. Podemos revisar los pesos calculados para este ejemplo:

```
print('Pesos : ',ppn.w_)
```

```
Pesos :  [-0.38375655 -0.70611756  1.83471828]
```

Para dibujar una línea que marque el límite de las clases, recordamos que el perceptrón realiza la operación:

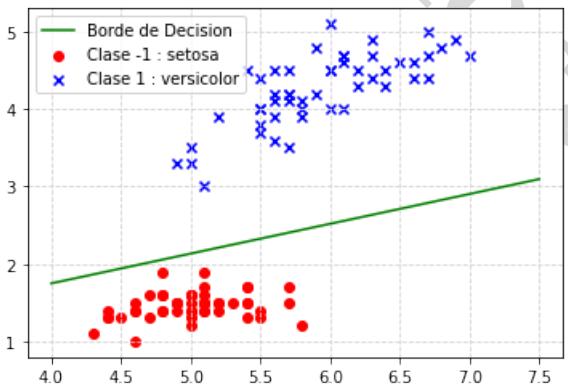
$$w_0 x_0 + w_1 x_1 + w_2 x_2$$

Cómo la característica de sesgo (*bias*) está definida como $x_0 = 1$, podemos despejar a x_2, obteniendo:

$$x_2 = -\frac{w_0 + w_1 x_1}{w_2}$$

Con este resultado, podemos mostrar los resultados en una gráfica:

```
# Borde de decision
x1 = np.linspace(4, 7.5, 2)
x2 = - (ppn.w_[0]+ppn.w_[1]*x1) / ppn.w_[2]
plt.plot(x1, x2, 'g', label = "Borde de Decision")
# Clase -1 : setosa
registros = y == -1
x1 = X[registros][:, 0]
x2 = X[registros][:, 1]
plt.scatter(x1, x2, c='r', marker='o', label="setosa")
# Clase 1 : versicolor
registros = y == 1
c1 = X[registros][:, 0]
c2 = X[registros][:, 1]
plt.scatter(c1, c2, c='b', marker='x', label="versicolor")
plt.legend()
plt.grid(color = 'lightgray', linestyle = '--')
```

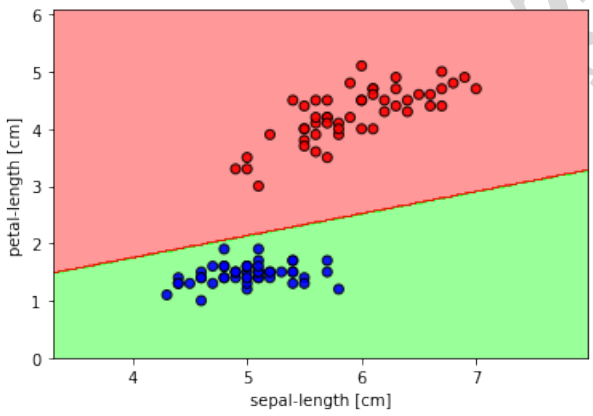


Por último, una función que ayude a graficar los resultados que, además, pone cada zona en un color diferente:

```python
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    colors = np.array(['lime', 'red', 'blue'])
    cmap = ListedColormap(colors[:len(np.unique(y))])
    # Superficies de decisión
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.4, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    # Conjuntos de cada clase
    plt.scatter(X[:,0],
                X[:,1],
                alpha=0.9,
                c=colors[y.astype(int)],
                edgecolor='black',)
```

Figure 1.16: Función para graficar las regiones

```
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('sepal-length [cm]')
plt.ylabel('petal-length [cm]')
plt.show()
```



Conociendo *scikit-learn*

scikit-learn (*sklearn*) es una biblioteca desarrollada en Python funciona sobre *NumPy*, *SciPy* y *matplotlib*; incluye muchos algoritmos de aprendizaje automático (*machine learning*) como los que revisaremos adelante en el curso.

El objetivo de esta sección es conocer su uso básico para poder desarrollar aplicaciones más avanzadas.

Conjuntos de datos (*datasets*)

Dentro de *datasets*, *sklearn* incluye varios conjuntos para pruebas; por ejemplo Iris:

```
from sklearn import datasets
import numpy as np

iris = datasets.load_iris()
X = iris.data[:, [2,3]]
y = iris.target[:]
print('Etiquetas : ',np.unique(y))
```

```
Etiquetas :  [0 1 2]
```

Partición en datos de entrenamiento y prueba

La función *train_ test_ split* de *model_ selection* genera particiones de datos para entrenamiento y prueba para determinar cómo se comporta con datos desconocidos:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1, stratify=y)
```

En este caso, tenemos 45 datos de prueba que corresponden con el 30 % y $105 = 70\,\%$ para entrenamiento. Podemos saber si nuestro conjunto de entrenamiento no está *sesgado*: es deseable que tenga la misma cantidad de datos de cada clase para que el algoritmo se comporte de buena forma; el parámetro *stratify* es el encargado de garantizar esta característica:

```
print('Total de etiquetas en y       :', np.bincount(y))
print('Total de etiquetas en y_train :', np.bincount(y_train))
print('Total de etiquetas en y_test  :', np.bincount(y_test))
```

```
Total de etiquetas en y       : [50 50 50]
Total de etiquetas en y_train : [35 35 35]
Total de etiquetas en y_test  : [15 15 15]
```

Escalamiento de características

Muchos algoritmos de aprendizaje requieren que los datos de entrada se encuentren escalados para un óptimo funcionamiento; dentro de *preprocessing*, está disponible la clase *StandardScaler*:

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

Perceptrón

Se encuentra incluído dentro del subpaquete *linear_ model* de *sklearn*; la mayoría de los modelos disponibles en *scikit-learn* soportan clasificación multiclase por omisión vía la técnica *One-versus-Rest* (*OvR*); con esto podemos utilizar las tres clases de flores:

```
from sklearn.linear_model import Perceptron
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
```

Podemos saber el total de predicciones equivocadas del modelo:

```
y_pred = ppn.predict(X_test_std)
print('Errores de clasificación : ',(y_test-y_pred).sum())
```

```
Errores de clasificación :  1
```

Además, muchos modelos incluyen un *score* para saber la exactitud del mismo:

```
print(Exactitud : ',ppn.score(X_test_std,y_test))
```

```
Exactitud :  0.9777777777777777
```

La misma medida está además disponible dentro de *metrics* como *accuracy_ score*:

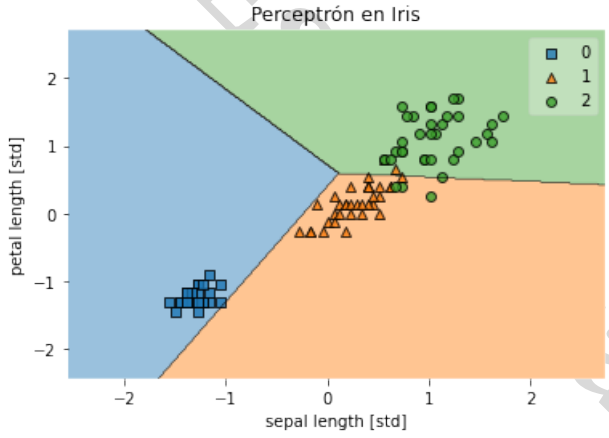
```
from sklearn.metrics import accuracy_score
print(Exactitud : ',accuracy_score(y_test,y_pred))
```

```
Exactitud :  0.9777777777777777
```

Gráfica de regiones de decisión

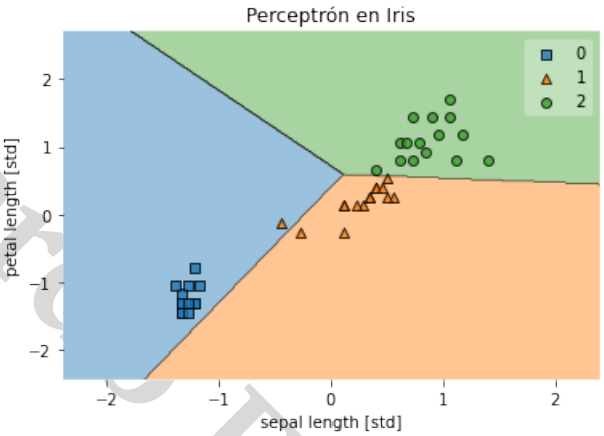
La función *plot_ decision_ regions* está disponible dentro de *mlxtend.plotting*, nos permite visualizar las clases y las líneas de separación que obtiene el modelo; para los datos de entrenamiento:

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.pyplot as plt

plot_decision_regions(X_train_std, y_train, clf=ppn)
plt.xlabel('sepal length [std]')
plt.ylabel('petal length [std]')
plt.title('Perceptrón en Iris')
plt.show()
```



Con los datos de prueba:

```
from mlxtend.plotting import plot_decision_regions
import matplotlib.pyplot as plt

plot_decision_regions(X_test_std, y_test, clf=ppn)
plt.xlabel('sepal length [std]')
plt.ylabel('petal length [std]')
plt.title('Perceptrón en Iris')
plt.show()
```



Regresión Logística: Probabilidades de clase

La regla del perceptrón es una buena forma de presentar los algoritmos de clasificación, pero tiene una gran desventaja: puede no converger si las clases no son linealmente separables. Intuitivamente, esto sucede porque siempre habrá al menos una muestra mal clasificada, provocando que los pesos se sigan actualizando, para resolver este inconveniente podría pensarse en probar con diferentes valores para la tasa de aprendizaje o aumentar el número de épocas. Pero existe una mejor forma: buscar un algoritmo alternativo; en este caso la *Logistic Regression* es simple, pero más poderoso para clasificación binaria.

Intuición y probabilidad condicional

Regresión logística (RL) es un modelo de clasificación sencillo, pero que se comporta bien en clases que no sean perfectamente separables linealmente, lo que lo convierte en uno de los algoritmos más comúnmente usados. Al igual que el perceptrón el algoritmo de regresión logística está implementado dentro de *sklearn* con el método *One-versus-Rest* para problemas multiclase.

Para presentar la idea detrás de RL como un modelo probabilístico, primero debemos ver el concepto de ***proporción de probabilidad*** (***odds ratio***): la probabilidad a favor de algún evento particular. Esta probabilidad puede escribirse como $\frac{p}{1-p}$ donde p es la probabilidad del evento positivo; es decir la probabilidad de que el evento que deseamos predecir suceda. Después, podemos definir la función ***logit*** como el logaritmo de la proporción de probabilidad (***log-odds***):

$$logit\left(p\right) = log\left(\frac{p}{1-p}\right)$$

Donde *log* es el logaritmo natural, convención adoptada en ciencias computacionales. La función *logit* recibe valores en el rango de 0 a 1 y los transforma en valores en todo el rango de los reales, que podemos utilizar para expresar una relación lineal entre los valores de las características y los *log-odds*:

$$logit\left(p\left(y=1|\mathbf{x}\right)\right) = w_0x_0 + w_1x_1 + \cdots + w_nx_n = \sum_{i=0}^{n} w_ix_i = \mathbf{w}^T\mathbf{x}$$

Aquí, $p\left(y=1|\mathbf{x}\right)$ es la probabilidad de que una muestra particular pertenezca a la clase 1 dadas sus características $\mathbf{x}$.

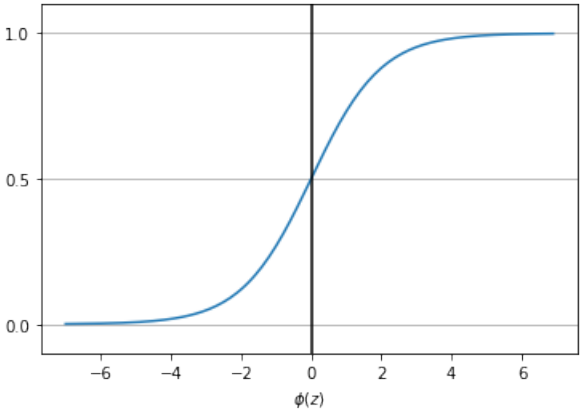
Finalmente, nos interesa predecir la probabilidad de que cierta muestra pertenezca a alguna clase particular, que es la forma inversa de la función *logit*. Esta función se conoce como *función logística sigmoide* o simplemente ***sigmoide*** caracterizada por su forma de S:

$$\phi\left(z\right) = \frac{1}{1+e^{-z}}$$

Donde z es la combinación lineal de los pesos y las características de la muestra:

$$z = w_0x_0 + w_1x_1 + \cdots + w_nx_n$$

```
import numpy as np
import matplotlib.pyplot as plt
# Sigmoide
def sigmoide(z):
  return 1.0/(1 + np.exp(-z))

z = np.arange(-7,7,0.1)
phi_z = sigmoide(z)
plt.plot(z, phi_z)
plt.axvline(0.0, color='k')
plt.ylim(-0.1,1.1)
plt.xlabel('z')
plt.ylabel('$\phi (z)$')
plt.yticks([0.0,0.5,1])
ax = plt.gca()
ax.yaxis.grid(True)
plt.show()
```



Con esto, el esquema de la regresión logística agrega la sigmoide como función de activación:

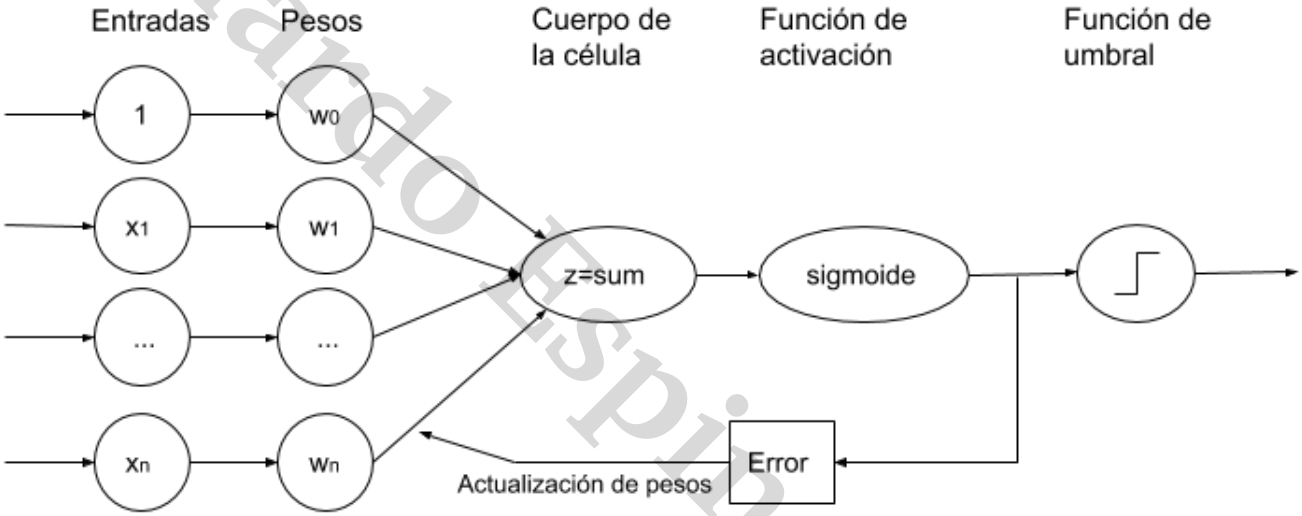


Figure 1.17: Esquema de regresión logística

Regresión Logística con sklearn

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(C=100.0, random_state=1)
lr.fit(X_train_std, y_train)
```

```
y_pred = lr.predict(X_test_std)
print('Errores de clasificación : ',(y_test-y_pred).sum())
print(Exactitud : ',lr.score(X_test_std,y_test))
```

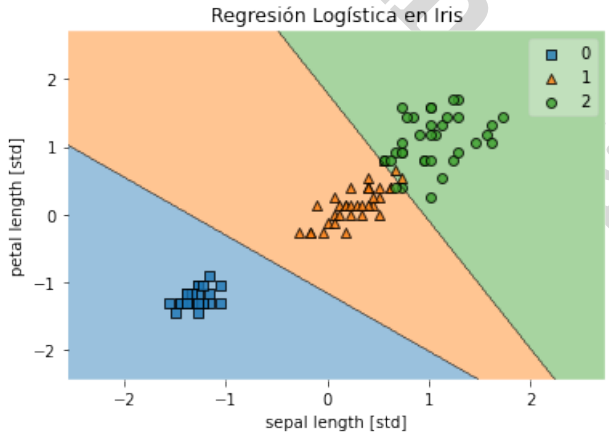
```
Errores de clasificación :  1
Exactitud :  0.9777777777777777
```

```
lr.coef_
```

```
array([[-6.93265988, -5.76495748],
       [-2.03192177, -0.03413691],
       [ 8.96458165,  5.79909439]])
```

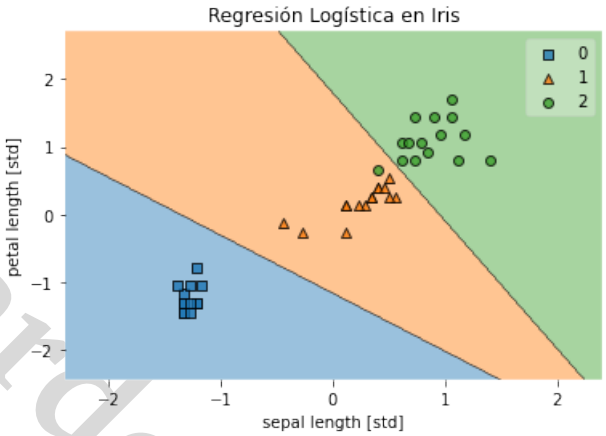
Gráfica del conjunto de entrenamiento:

```
# Regiones de decisión
plot_decision_regions(X_train_std, y_train, clf=lr)

plt.xlabel('sepal length [std]')
plt.ylabel('petal length [std]')
plt.title('Regresión Logística en Iris')
plt.show()
```



Gráfica del conjunto de pruebas:

```
plot_decision_regions(X_test_std, y_test, clf=lr)
plt.xlabel('sepal length [std]')
plt.ylabel('petal length [std]')
plt.title('Regresión Logística en Iris')
plt.show()
```



Elipses como separación de clases

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
```

```
data = pd.read_csv('https://bit.ly/3BijEAo',header=0)
data.head()
X = data.iloc[:,:-1].values
y = data.iloc[:,-1].values
```

```
color = ['r','b']
for i,c in enumerate(color):
  registros = y == i
  x1 = X[registros][:,0]
  x2 = X[registros][:,1]
  plt.scatter(x1,x2,c=c,label='Clase '+str(i))
plt.legend()
plt.grid(True)
plt.show()
```

```
lr = LogisticRegression()
# Elipse a x1**2 + b x2**2 + c = 0
#       w1 x1**2 + w2 x2**2 + w0 = 0
# x2 = +/- sqrt ( (w0 + w1 x1**2) / w2 )
lr.fit(X**2,y)
w_ = [*lr.intercept_, *lr.coef_[0]]
w_
```

```
x1=np.linspace( -(-w_[0]/w_[1])**(1/2), (-w_[0]/w_[1])**(1/2), 10000 )
x2 = ( (w_[0] + w_[1]*x1**2) / (-w_[2]) ) ** (1/2)
x2n = -x2
plt.plot(x1,x2, 'g')
plt.plot(x1,x2n, 'g')
color = ['r','b']
for i,c in enumerate(color):
  registros = y == i
  x1 = X[registros][:,0]
  x2 = X[registros][:,1]
  plt.scatter(x1,x2,c=c,label='Clase '+str(i))
plt.legend()
plt.grid(True)
plt.show()
```