1. Aprendizaje supervisado

En el *aprendizaje supervisado*, un supervisor lleva a cabo algunas etapas en la construcción del modelo: determinación de las clases, elección y prueba de las características discriminantes, selección de la muestra, cálculo de funciones discriminantes y evaluación del resultado.

1.1. Regresión lineal

La regresión lineal es una técnica de modelado estadístico que se emplea para describir una variable de respuesta continua como una función de una o varias variables predictoras.

Es un área bien desarrollada y conocida de la estadística; la ubicuidad de sus métodos en el análisis de datos surge de la facilidad con la que se pueden ajustar modelos para describir las características principales de un proceso o una población. Además de ser útil para la descripción, también es muy útil para la predicción ya que los modelos obtenidos en general proveen buenas aproximaciones de relaciones complejas.

1.1.1. Regresión lineal simple

En éste, se tiene una sola variable predictora (x) usada para predecir el valor de la variable de respuesta (y).

La representación matemática es:

$$\hat{y} = a + bx,$$

$$a = \bar{y} - b\bar{x},$$

$$b = \frac{s_{xy}}{s_x^2},$$

donde:

♢ $\bar{y}$ y $\bar{x}$ son las medias de y y x respectivamente

♢ s_x^2 es la varianza de x

♢ s_{xy} es la covarianza de y y x

Coeficiente de correlación lineal

Se obtiene con:

$$r = \frac{s_{xy}}{s_x s_y},$$

♢ s_x es la desviación estándar de x

♢ s_y es la desviación estándar de y

♢ Varía entre -1 y 1

1.1.2. Regresión lineal multivariada

En el marco de la regresión lineal, los datos pueden verse como n parejas: $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$. La visión estadística supone que $y_i, i = 1, 2, \ldots, n$ provienen de una variable aleatoria, identificada como la variable de respuesta Y_i y que el vector explicativo que la acompaña $\mathbf{x}_i$ no es aleatorio y puede usarse para explicar y_i. En ocasiones, el objetivo estadístico es predecir un valor no observado y_0 a partir de un vector observado $\mathbf{x}_0$ usando una función de predicción obtenida a partir de los datos. En estadística se predicen valores de una variable aleatoria y se estiman los parámetros que describen una distribución de la misma.

Las variables $\mathbf{x}_i$, y_i son intrínsecamente diferentes; el objetivo es describir o modelar el valor esperado de la variable Y_i mientras que $\mathbf{x}_i$ es un vector de interés secundario. Comúnmente se asume que el vector explicativo no tiene una distribución, en cambio contiene valores fijos medidos sin error o en control absoluto del investigador. Esta suposición puede resultar engañosa y la regresión lineal es un método valioso para obtener información sobre el proceso o la población de la que provienen los datos. El vector $\mathbf{x}$ se conoce como *explicativo* o *predictor*, dependiendo del objetivo particular del análisis; por simplicidad se utilizará el término *vector predictor* sin importar el tipo de análisis que se esté realizando.

El modelo de regresión lineal especifica que el valor esperado de Y_i es una función lineal de $\mathbf{x}_i$ dado por:

$$E\left(Y_i|x_i\right) = \beta_0 + \beta_1 x_{i,1} + \ldots + \beta_p x_{i,p}$$

O en su versión vectorial:

$$E\left(Y_i|x_i\right) = \mathbf{x}_i^T \boldsymbol{\beta}$$

donde $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_p]^T$ es un vector de constantes desconocidas. El vector $\mathbf{x}_i = [1, x_{i,1}, \ldots, x_{i,p}]$ contiene las observaciones de los p predictores variables; la longitud de los vectores $\boldsymbol{\beta}$ y $\mathbf{x}_i$ es $q = p + 1$.

La aplicación de la predicción es simple en el sentido de que los detalles del modelo predictivo no tienen especial importancia si produce predicciones precisas de Y; es más común que se tenga interés en entender la influencia de cada variable predictora sobre Y, más precisamente sobre el valor esperado de Y.

Visualización interactiva: https://bit.ly/486nY32.

1.1.2.1. Mínimos cuadrados

Explicación en Prometeo: https://bit.ly/3AYiwPN.

La primera tarea computacional del análisis de regresión lineal es calcular un estimado del vector de parámetros $\boldsymbol{\beta}$. En estadística y ciencia de datos, el estimador de mínimos cuadrados casi siempre es la primera elección porque es fácil de entender y calcular. Aún más, su precisión compite con la de una gran variedad de métodos más complejos; los intervalos de confianza y las pruebas de hipótesis con respecto a los parámetros $\beta_0, \beta_1, \ldots, \beta_p$ son sencillos.

El objetivo de los mínimos cuadrados es minimizar la suma de los residuos al cuadrado; los residuos son las diferencias entre el valor observado y_i y los valores ajustados $\hat{y}_i = \mathbf{x}_i^T \hat{\boldsymbol{\beta}}, i = 1, \ldots, n$, donde $\hat{\boldsymbol{\beta}}$ es un vector de números reales de longitud q. La suma de los residuos al cuadrado se calcula con:

$$\begin{aligned} S\left(\hat{\boldsymbol{\beta}}\right) &= \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2 \\ &= \sum_{i=1}^{n} \left(y_i - \mathbf{x}_i^T \boldsymbol{\beta}\right)^2 \end{aligned}$$

Se minimiza con la elección de $\boldsymbol{\beta}$; al tener valor desconocido necesitamos calcularlo para poder continuar. El vector $\hat{\boldsymbol{\beta}}$ que minimiza $S(\cdot)$ se conoce como el estimador de mínimos cuadrados y, por definición, cuaquier otro vector entrega una suma de errores al cuadrado al menos tan grande como este estimador. Una derivación puede consultarse en https://bit.ly/2KfhAyl. Lo importante desde nuestra perspectiva es que el estimador $\boldsymbol{\beta}$ es la solución de las *ecuación normal*:

$$\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}$$

$\underset{n\times q}{\mathbf{X}}$ se forma al *apilar* los vectores predictivos $\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T$. El vector $\mathbf{y} = [y_1, \ldots, y_n]^T$ consiste de n productos de las variables aleatorias $Y_1, \ldots, Y_n$. Si $\mathbf{X}^T\mathbf{X}$ es invertible, entonces la solución de la ecuación normal es:

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

1.1.2.2. Regularización Un modelo así obtenido funciona muy bien si el número de muestras (n) es mucho mayor que el número de parámetros (p):

$$n \gg p \Rightarrow \textit{poca varianza}$$

Cuando esta condición no se cumple, se puede presentar alta correlación entre las variables predictoras, provocando que el modelo esté *sobreajustado* debido aque el modelo es muy complejo. Es decir, nuestra solución que funciona muy bien para los datos de entrenamiento pero muy mal para datos nuevos.

En general, cuando usamos regularización, se utiliza el error cuadrático medio (MSE) como función de costo, añadiendo un término que penaliza la complejidad del modelo:

$$J = MSE + \alpha C,$$

C es la medida de complejidad del modelo. Dependiendo de cómo se mide la complejidad, se tienen distintos tipos de regularización. El hiperparámetro α indica qué tan importante es que el modelo sea simple en relación a qué tan importante es su rendimiento. Cuando se utiliza regularización, se reduce la complejidad del modelo al mismo tiempo que se minimiza la función de costo. Existen dos formas básicas: Lasso ($L1$) y Ridge ($L2$).

Regularización *Lasso* ($L1$) En esta regularización, la complejidad C se mide como la media del valor absoluto de los coeficientes del modelo. Esto se puede aplicar a regresión lineal, polinómica, regresión, redes neuronales, máquinas de soporte vectorial, etc. Matemáticamente se define como:

$$C = \frac{1}{n}\sum_{i=1}^{n}|w_1|,$$

con esto, para el caso del MSE, la función de costo queda:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha\frac{1}{n}\sum_{i=1}^{n}|w_1|$$

Lasso es útil cuando se sospecha que varios de los atributos de entrada (*características*) son irrelevantes. Al usar Lasso, se fomenta que la solución sea poco densa, es decir, se favorece que algunos de los coeficientes tengan valor 0. Esto puede ser útil para descubrir cuáles de los atributos de entrada son relevantes y, en general, para obtener un modelo que generalice mejor. Lasso funciona mejor cuando los atributos no están muy correlacionados entre ellos.

Regularización *Ridge* ($L2$) En este tipo de regularización, la complejidad C se mide como la media del cuadrado de los coeficientes del modelo. También se puede aplicar a diversas técnicas de aprendizaje automático. Matemáticamente se define como:

$$C = \frac{1}{2n} \sum_{i=1}^{n} w_1^2,$$

y la función de costo:

$$J = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \alpha \frac{1}{2n} \sum_{i=1}^{n} w_1^2$$

Ridge funciona bien cuando se sospecha que varios de los atributos de entrada están correlacionados entre ellos. Ridge hace que los coeficientes obtenidos sean más pequeños, esta disminución de los coeficientes minimiza el efecto de la correlación entre los atributos de entrada y hace que el modelo generalice mejor. Ridge funciona mejor cuando la mayoría de los atributos son relevantes.

Regularización *ElasticNet* ($L1$ y $L2$) Combina las regularizaciones $L1$ y $L2$; con el parámetro r podemos indicar que importancia relativa tienen Lasso y Ridge respectivamente. Matemáticamente:

$$C = r \times Lasso + (1 - r) \times Ridge,$$

desarrollado para el caso del error cuadrático medio, se obtiene:

$$J = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + r \times \alpha \frac{1}{n} \sum_{i=1}^{n} |w_1| + (1 - r) \times \alpha \frac{1}{2n} \sum_{i=1}^{n} w_1^2$$

ElasticNet se recomienda cuando se cuenta con gran número de características: algunos serán irrelevantes y otros estarán correlacionados.

Nota *scikit-learn* ofrece el hiperparámetro *"penalty"* en muchos de sus modelos; se puede utilizar "$l1$" o "$l2$" en *penalty* para elegir la regularización a usar.

1.2. Regresión polinomial

En ocasiones una linea recta no es la mejor forma de hacer regresión sobre un conjunto de datos; es común que la relación entre las características y la variable de respuesta (objetivo) no se puede describir correctamente con una línea recta. Es posible que un polinomio funcione mejor para realizar las predicciones de la variable buscada.

Un polinomio puede expresarse de la siguiente manera:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 \ldots + \beta_n x^n$$

Donde:

♢ y es la variable de respuesta que se desea predecir.

♢ x es la característica utilizada para realizar la predicción.

♢ β_0 es el término independiente.

♢ n es el grado del polinomio.

♢ $\beta_1 \ldots, \beta_n$ son los coeficientes que deseamos estimar al ajustar el modelo con los valores de x, y disponibles.

Si comparamos la expresión de la regresión polinomial con la usada para regresión lineal:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \ldots + \beta_n x_n$$

Notación que son muy similares, esto no es una coincidencia: **la regresión polinomial es un modelo lineal** usado para describir relaciones no lineales; ls *magia* recae en crear nuevas características elevando las características originales a alguna potencia.

Por ejemplo, si tenemos una característica x y deseamos un polinomio de tercer grado, la regresión incluirá tanto a x^2 como a x^3 para la estimación de los coeficientes.

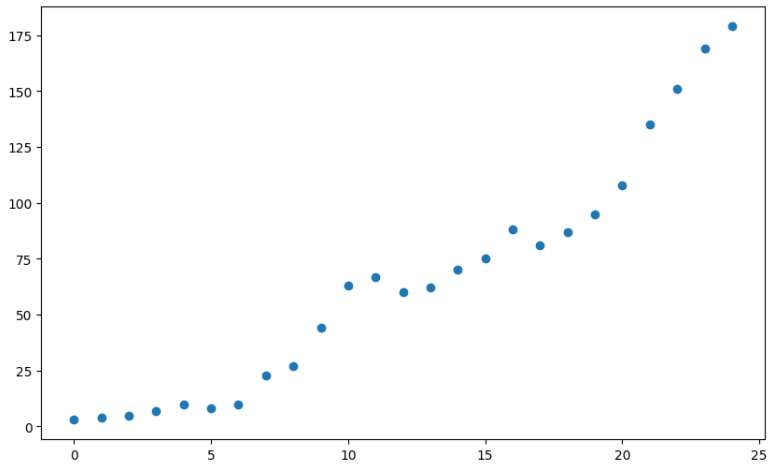
Ejemplo.

Bibliotecas:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Conjunto de datos:

```
y = [3, 4, 5, 7, 10, 8, 10, 23, 27, 44, 63, 67, 60, 62, 70, 75, 88, 81,
     87, 95, 108, 135, 151, 169, 179]
x = np.arange(len(y))
plt.figure(figsize=(10,6))
plt.scatter(x, y)
plt.show()
```



Vemos que no parece funcionar bien una regresión lineal y buscamos una curva que describa mejor la relación. En particular, opdemos inferir que un polinomio de grado 2. Entonces nuestro modelo es:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2$$

Para crear las características polinomiales, existe *PolynomialFeatures* dentro de *sklearn*:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=False)
```

include_bias debe tener valos *false* porque la regresión lineal se encargará del término independiente.

Ajustamos y transformamos el objeto *poly*:

```
poly_features = poly.fit_transform(x.reshape(-1, 1))
poly_features
```

reshape$(-1, 1)$ trnasforma el arreglo de 1D a 2D, necesario para el ajuste. Al observar la salida vemos que el segundo elemento son las características originales elevadas al cuadrado, tal como lo deseamos.

Creamos el modelo lineal:

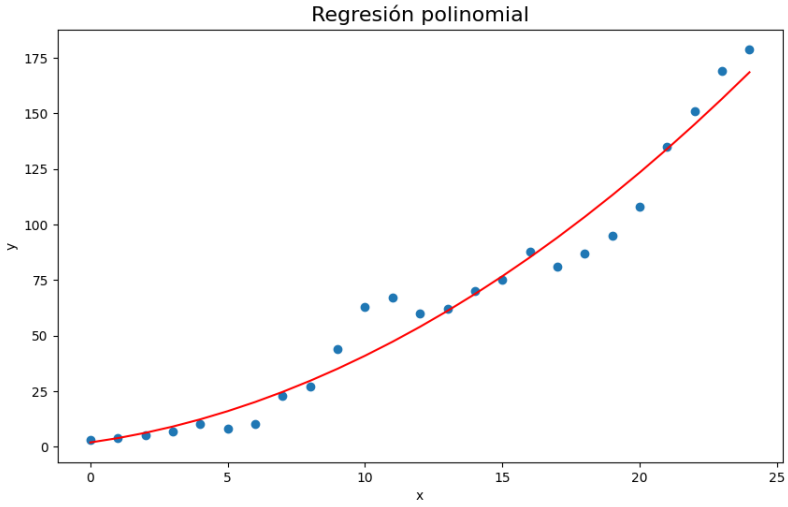
```
from sklearn.linear_model import LinearRegression
poly_reg_model = LinearRegression()
```

Ajustamos y predecimos los valores:

```
poly_reg_model.fit(poly_features, y)
y_pred = poly_reg_model.predict(poly_features)
```

Graficamos el resultado:

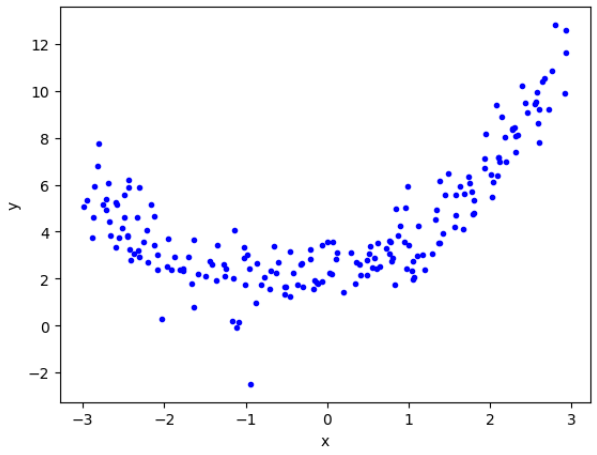
```
plt.figure(figsize=(10, 6))
plt.scatter(x, y)
plt.plot(x, y_pred, c="red")
plt.title('Regresión polinomial', size=16)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



1.2.1. Determinar el grado del polinomio

Generamos un nuevo conjunto de datos con 200 muestras:

```
import numpy as np
import matplotlib.pyplot as plt
x = 6 * np.random.rand(200, 1) - 3
y = 0.8*x**2 + 0.9*x + 2 + np.random.randn(200, 1)
plt.plot(x, y, 'b.')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Importamos otros elementos útiles:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import r2_score, mean_squared_error
```

Separamos el conjunto de entrenamiento y pruebas:

```
x_train,x_test,y_train,y_test=train_test_split(X, y, test_size=0.2,
                                                random_state=2)
```

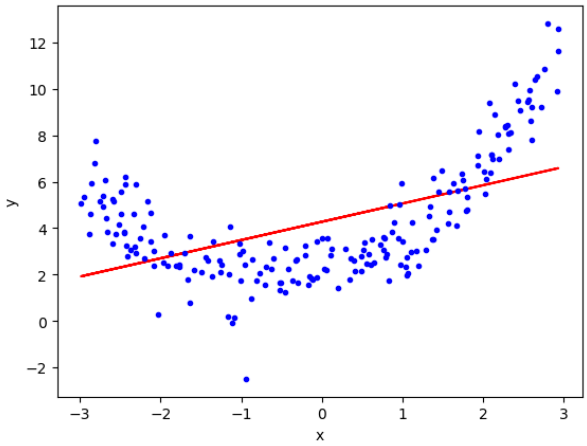
Intentamos con un modelo lineal y revisamos sus métricas:

```
lr = LinearRegression()
lr.fit(x_train, y_train)
y_pred = lr.predict(x_test)
print('  R2 : ',r2_score(y_test, y_pred))
print('RMSE : ',np.sqrt(mean_squared_error(y_test, y_pred)))
```

```
  R2 :  0.316195321755792
RMSE :  2.327275341098332
```

Ambos resultados son malos, esto indica que el modelo lineal no es bueno, podemos incluso graficar para corroborarlo.

```
plt.plot(x_train, lr.predict(x_train), color='r')
plt.plot(x, y, 'b.')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Es momento de probar con una regresión polinomial de grado 2.
Generamos las características de entrenamiento y prueba para el modelo:

```
poly = PolynomialFeatures(degree=2, include_bias=False)
x_train_poly = poly.fit_transform(x_train)
x_test_poly = poly.transform(x_test)
```

Entrenamos el modelo lineal y calculamos las métricas:

```
lr = LinearRegression()
lr.fit(x_train_poly, y_train)
y_pred = lr.predict(x_test_poly)
print('  R2 : ',r2_score(y_test, y_pred))
print('RMSE : ',np.sqrt(mean_squared_error(y_test, y_pred)))
```

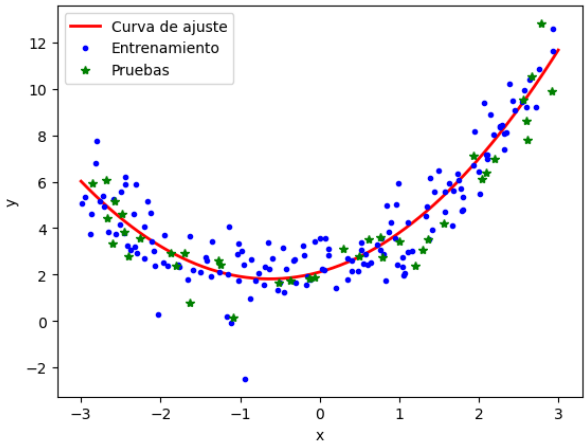
```
  R2 :  0.8797492410737734
RMSE :  0.9759456823533897
```

Observamos que ambas métricas son significativamente mejores que las de la regresión lineal, podemos obtener los coeficientes y el término independiente de la siguiente manera:

```
print(lr.coef_)
print(lr.intercept_)
```

Además de mostrar gráficamente la curva de ajuste junto con los puntos de entrenamiento y prueba:

```
x_new = np.linspace(-3, 3, 200).reshape(200, 1)
x_new_poly = poly.transform(x_new)
y_new_pred = lr.predict(x_new_poly)
plt.plot(x_new, y_new, "r", linewidth=2, label='Curva de ajuste')
plt.plot(x_train, y_train, "b.",label='Entrenamiento')
plt.plot(x_test, y_test, "g*",label='Pruebas')
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```



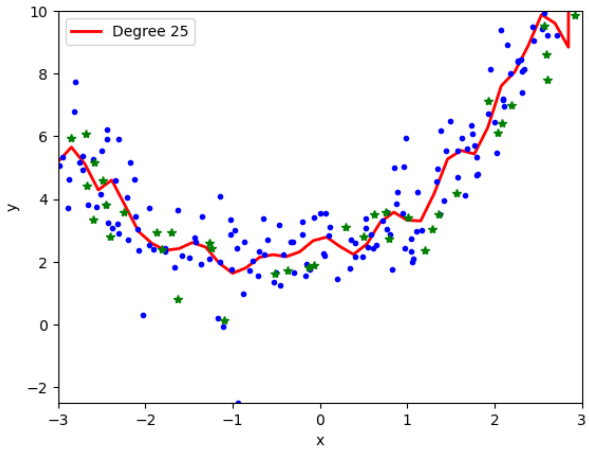
En los ejemplos presentados hemos utilizado grado 2 debido a que sabíamos *a priori* que es el grado que mejor se ajusta a los datos, pero cuando no es así, se debe buscar probando diferentes valores y revisando sus métricas y, de ser posible, con ayuda de una visualización.

La siguiente función ayuda para generar la gráfica y devuelve las métricas; recibe como parámetro los conjuntos de entrenamiento y pruebas a utilizar:

```
from sklearn.pipeline import make_pipeline
def polynomial_regression(degree,x_train,x_test,y_train,y_test,plot=False):
  test_size = y_test.shape[0]
  x_new=np.linspace(-3, 3, test_size).reshape(test_size, 1)
  pipe_poly = make_pipeline(
    PolynomialFeatures(degree=degree, include_bias=False),
    LinearRegression(),
   )
  pipe_poly.fit(x_train, y_train)
  y_new_pred = pipe_poly.predict(x_new)
  if plot:
    plt.plot(x_new, y_new_pred,'r',label='Degree '+str(degree),linewidth=2)
    plt.plot(x_train, y_train, 'b.')
    plt.plot(x_test, y_test, 'g*')
    plt.legend(loc='upper left')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.axis([-3, 3, -2.5, 10])
    plt.show()
  r2 = r2_score(y_test, y_new_pred)
  rmse = np.sqrt(mean_squared_error(y_test, y_new_pred))
  return r2, rmse
```

Se pueden probar diferentes grados para el polinomio, es el primero parámetro de la función, por ejemplo, con un polinomio de grado 25, se obtiene:

```
polynomial_regression(25,x_train,x_test,y_train,y_test,plot=True)
```




```
(-1.8844933950353555, 4.7443414819507295)
```

1.2.2. Regresión polinomial con múltiples características

Es muy común que los conjuntos de datos contengan más de una característica para predecir la variable objetivo. Podemos simular esta situación para dos características.

Bibliotecas:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

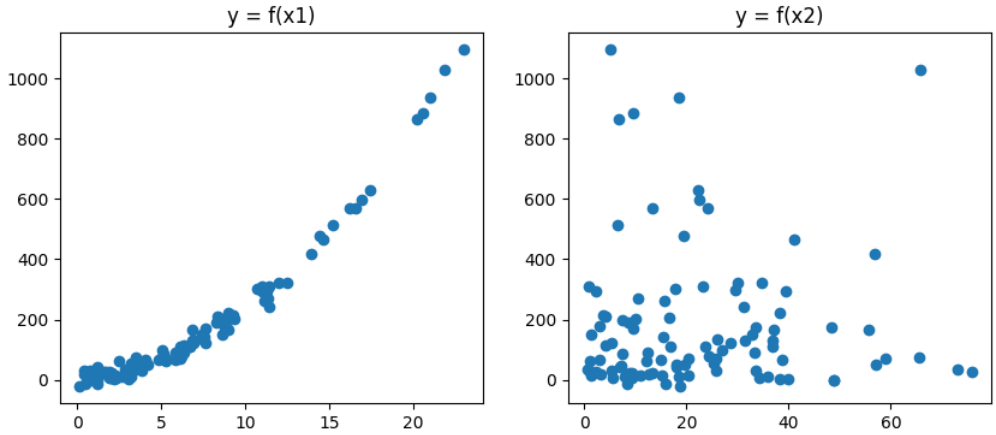
Datos artificiales:

```
np.random.seed(1)
x1 = np.absolute(np.random.randn(100, 1) * 10)
x2 = np.absolute(np.random.randn(100, 1) * 30)
y = 2*x1**2 + 3*x1 + 2 + np.random.randn(100, 1)*20
```

Tenemos 100 valores para cada variable independiente x_1, x_2, así como 100 valores para la variable objetivo y dependientes de x_1 y x_2; todas ellas con ruido *aleatorio*.

Gráfica de y contra cada una de las variables:

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
axes[0].scatter(x1, y)
axes[1].scatter(x2, y)
axes[0].set_title("y = f(x1)")
axes[1].set_title("y = f(x2)")
plt.show()
```



Convertimos el conjunto en un *DataFrame*:

```
df = pd.DataFrame({'x1':x1.reshape(100,), 'x2':x2.reshape(100,),
                   'y':y.reshape(100,)}, index=range(0,100))
df.head()
```

Separamos conjuntos para entrenamiento y pruebas:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
X, y = df[['x1', 'x2']], df['y']
poly = PolynomialFeatures(degree=2, include_bias=False)
poly_features = poly.fit_transform(X)
X_train,X_test,y_train,y_test=train_test_split(poly_features,y,
                                                test_size=0.3,random_state=42)
```

El modelo polinomial de grado 2 para las variables x_2 y x_2 queda de la siguiente manera:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1^2 + \beta_4 x_2^2 + \beta_5 x_1 x_2$$

Esto se debe a que *poly.fit_transform*(X) añadió tres características nuevas al conjunto, x_2^2 y x_2^2 ya eran esperadas y no requieren explicación; en cambio el término $x_1 x_2$ se conoce como *término de interacción* y es importante porque una variable podría ser dependeinte de la otra; el modelo *PolynomialFeatures* lo genera de forma automática.

Creamos el modelo lineal:

```
poly_reg_model = LinearRegression()
poly_reg_model.fit(X_train, y_train)
```

Verificamos el comportamiento de nuestra regresión:

```
poly_reg_model.fit(X_train, y_train)
y_pred = poly_reg_model.predict(X_test)
from sklearn.metrics import mean_squared_error
poly_reg_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
poly_reg_rmse
```

```
20.937707839078772
```

Calculamos la raíz de *mean_squared_error* para obtener la métrica conocida como *RMSE* (*Root Mean Square Error*); esta medida indica qué tan lejos están los valores predichos por el modelo (*y_pred*) de los valores reales (*y_test*). En términos generales, entre más pequeño se el valor de *RMSE*, mejor será el modelo.

Obtengamos un modelo de regresión lineal puro.

```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,
                                                random_state=42)
lin_reg_model = LinearRegression()
lin_reg_model.fit(X_train, y_train)
y_pred = lin_reg_model.predict(X_test)
lin_reg_rmse = np.sqrt(mean_squared_error(y_test, y_pred))
lin_reg_rmse
```

```
62.302487453878506
```

Comparando ambos valores del RMSE, vemos que el modelo polinomial es 3 veces mejor que un lineal.

1.3. Perceptrón

1.3.1. En 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.

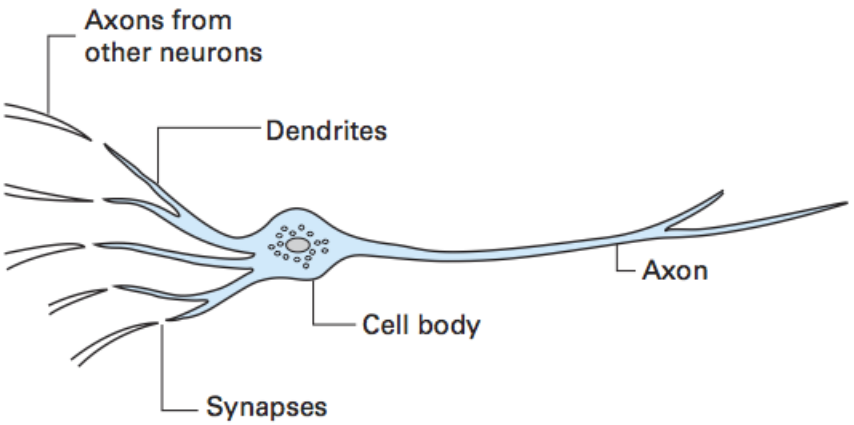


Figure 1: Esquema de una neurona biológica

1.3.2. 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*.

1.3.2.1. 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:

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

1.3.2.2. 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.

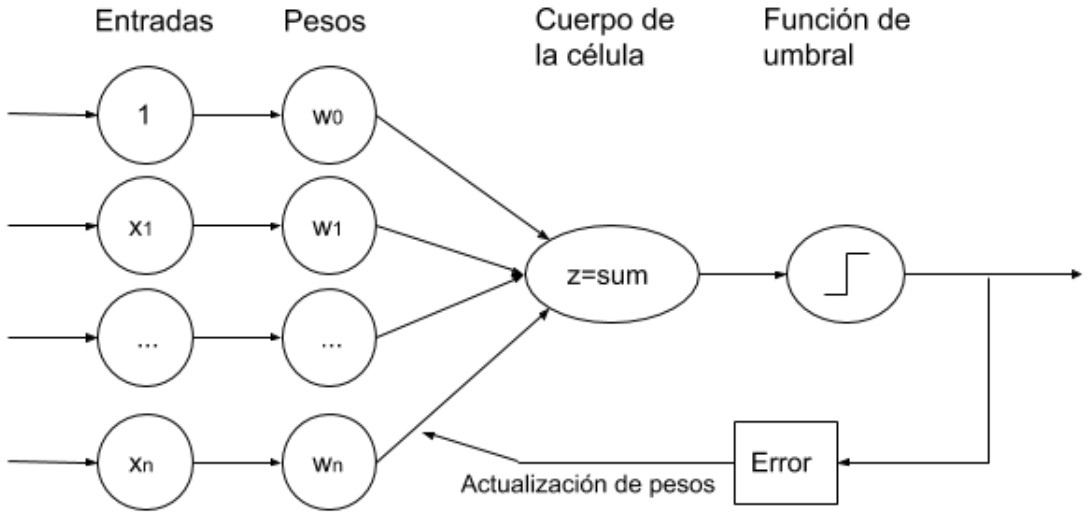


Figure 2: Esquema del perceptrón de Rosenblatt

La regla del perceptrón de Rosenblatt es bastante simple y puede resumirse en los pasos del algoritmo 1.

Algorithm 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 3: Actualizaciones de los pesos en el ejemplo

*

Implementación desde cero del Perceptrón

Realizaremos una implementación orientada a objetos de este algoritmo

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]
```

1.3.3. 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[1]. 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

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

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.

```
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()
```

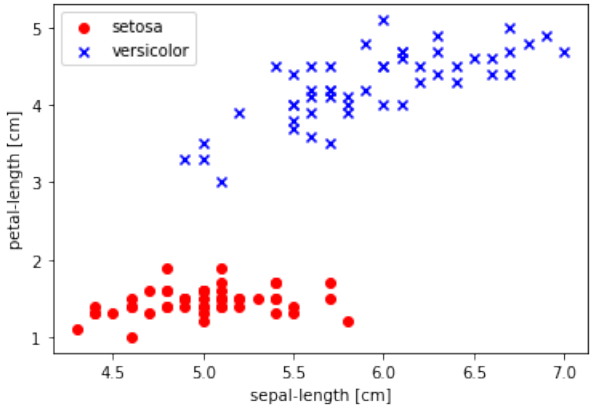
	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

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:

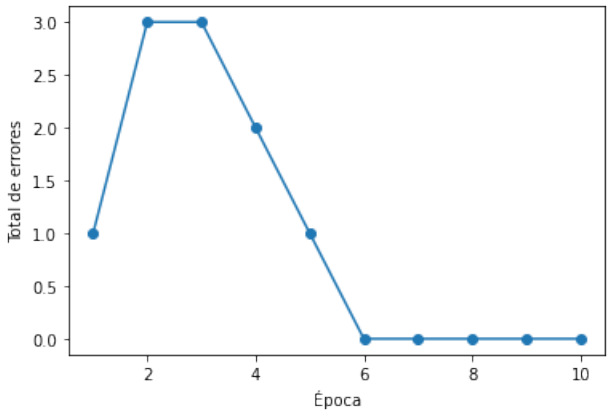
```
import numpy as np

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 encontrando una frontera de decisión que separe las dos clase:



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:

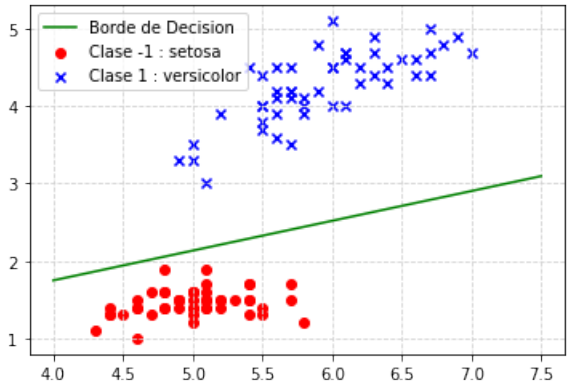
$$w_0 x_0 + w_1 x_1 + w_2 x_2$$

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 = '--')
```



Ejemplo con scikit-learn

1.4. Evaluación de modelos

Existen diferentes formas de evaluar modelos supervisados según su funcionalidad, por ejemplo, para problemas de regresión, se puede usar el valor R^2 o el $RSME$; en cambio para problemas de clasificación se suelen utilizar otras métricas.

En las secciones anteriores, hemos evaluado el rendimiento de los modelos con ayuda de su *exactitud* (*accuracy*) que es una métrica útil con la que se puede cuantificar el rendimiento general de un modelo de clasificación. Sin embargo, existen otras métricas del rendimiento que pueden usarse para determinar la relevancia de un modelo, tales como *precision* (precisión), sensitividad (*recall*) y *F1-score*.

1.4.1. Revisión de varias métricas de evaluación de rendimiento

Antes de entrar a los detalles de las métricas, revisemos la *matriz de confusión*, es una matriz que despliega el rendimiento de un algoritmo de aprendizaje: es una matriz cuadrada donde se reportan los totales de las predicciones *verdaderas positivas* (TP), *verdaderas negativas* (TN), *falsas positivas* (FP) y *falsas negativas* (FN) de un clasificador, como se muestra en la figura 4.

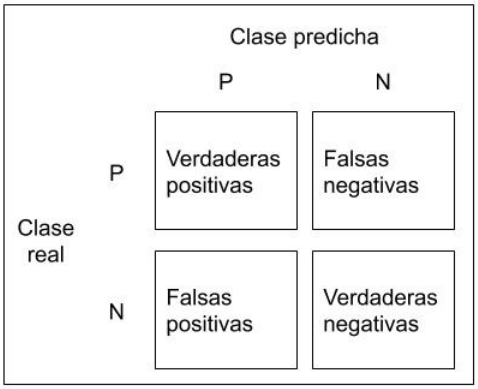


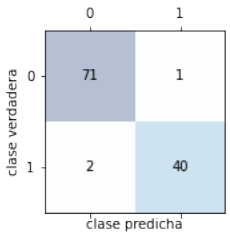
Figura 4: Posibles casos para las predicciones

Dentro de *scikit-learn* existe la función *confusion_ matrix*:

```
[[71  1]
 [ 2 40]]
```

Y también es posible producir una imagen con esta información con ayuda de *matshow* incluída dentro de *matplotlib*:

```
fig, ax = plt.subplots(figsize=(2.5, 2.5))
ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
for i in range(confmat.shape[0]):
  for j in range(confmat.shape[1]):
    ax.text(x=j, y=i, s=confmat[i,j])
```



Puede resultar más fácil interpretar los valores obtenidos y con ellos calcular varias métricas de error.

1.4.1.1. Métricas de *precision* y *recall* de un modelo de clasificación Tanto el *error* (*ERR*) como la exactitud (*accuracy ACC*) de la predicción proveen información general sobre cuantas muestras fueron clasificadas erróneamente. El error se define como la suma de predicciones falsas entre el número total de predicciones:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

La exactitud (*accuracy*) de la predicción puede calcularse directamente a partir del error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

La *proporción de verdaderas positivas* (*TPR*) y la *proporción de falsas positivas* (*FPR*) son métricas de rendimiento especialmente útiles en problemas de clases desbalanceadas:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}; \quad TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

Por ejemplo, al diagnosticar tumores nos interesa más detectar tumores malignos para poder ayudar con el tratamiento apropiado para el paciente. Sin embargo, también es importante disminuir el número de tumores benignos que fueron erróneamente clasificados como malignos (FP) para evitar preocupar innecesariamente a algún paciente. En contraste a la FPR, la TPR provee información útil sobre la fracción de muestras positivas que fueron identificadas correctamente de entre el total de positivas (P).

Las métricas precisión (*precision PRE*) y sensitividad (*recall REC*) se relacionan con las proporciones positivas y negativas; de hecho, REC es un sinónimo de TPR:

$$PRE = \frac{TP}{TP + FP}; \quad REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

En la práctica, la combinación de PRE y REC para obtener la métrica llamada $F1 - score$:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

Estas métricas se encuentran implementadas en *scikit-learn* dentro del módulo *sklearn.metrics*:

```
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score, f1_score

print('Precisión : %.3f' % precision_score(y_true=y_test, y_pred=y_pred))
print('   Recall : %.3f' % recall_score(y_true=y_test, y_pred=y_pred))
print('       F1 : %.3f' % f1_score(y_true=y_test, y_pred=y_pred))
```