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.

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.

1.1.3. Ejemplo con Python

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; la *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.

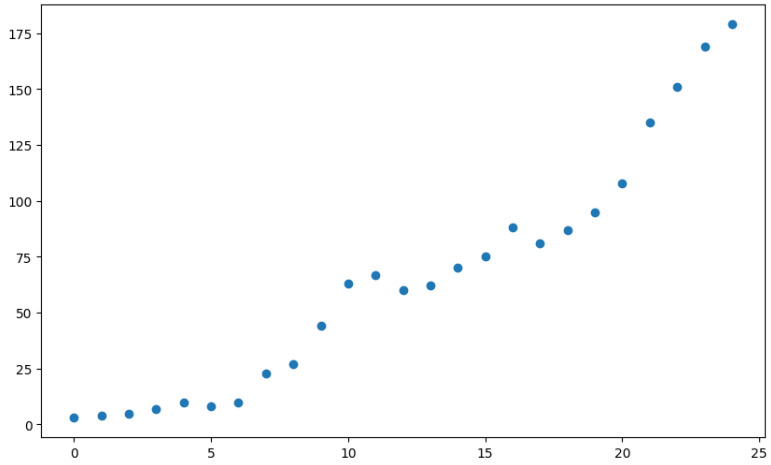
Ejemplo.

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:

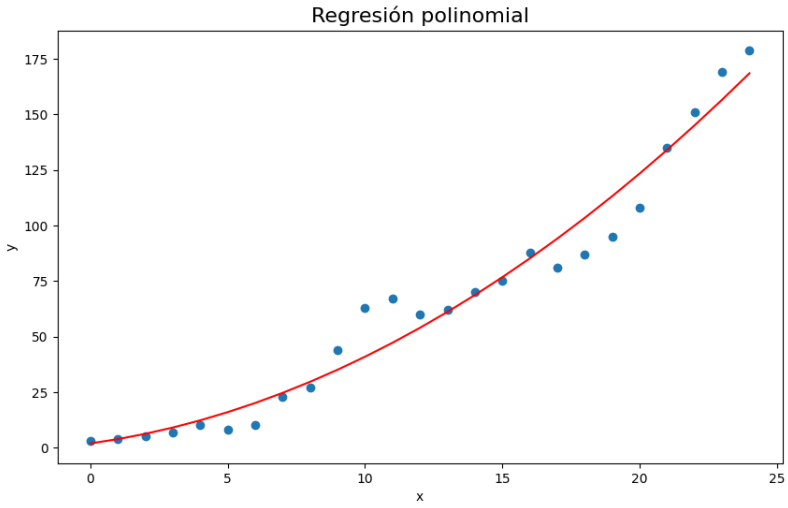
```
from sklearn.linear_model import LinearRegression
poly_reg_model = LinearRegression()
```

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()
```



1.3. Perceptrón

1.3.1. 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.

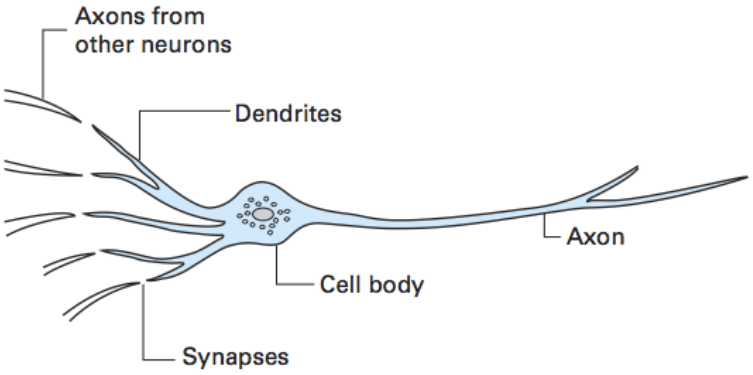


Figure 1: Esquema de una neurona biológica

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*.

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:

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

1.3.2.1. 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.

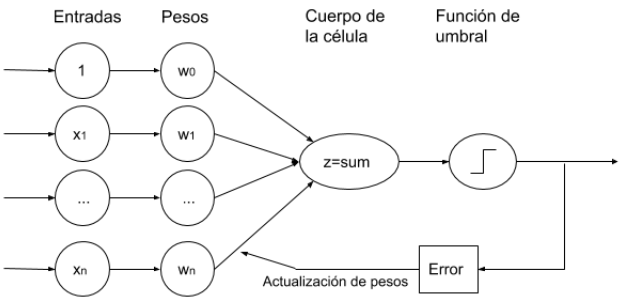


Figure 2: Esquema del perceptrón de Rosenblatt

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

*

Ejemplo

Conjunto de datos *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))
```

Conjuntos de entrenamientos y pruebas:

```
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))
```

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)
```

El preceptrón 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())
```

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

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

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

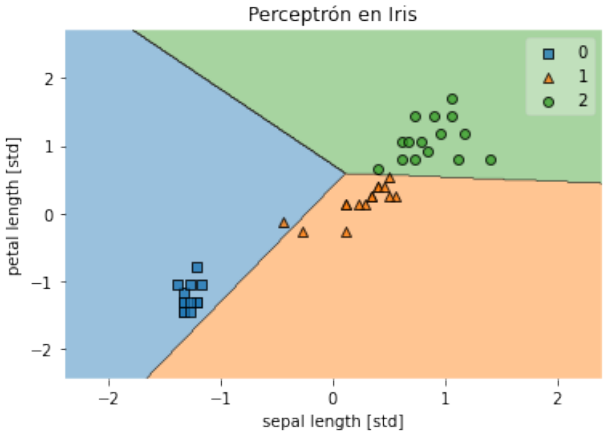
```
from sklearn.metrics import accuracy_score
print(Exactitud : ',accuracy_score(y_test,y_pred))
```

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:

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()
```



1.4. Regresión logística: Probabilidades de clase

1.4.1. 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, $p(y=1|\mathbf{x})$ 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:

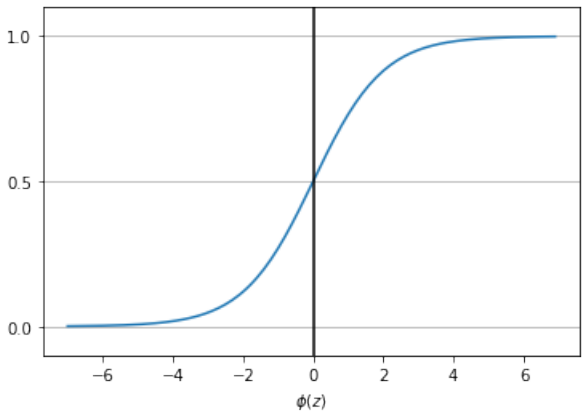
$$\phi(z) = \frac{1}{1+e^{-z}}$$

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

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_n x_n$$

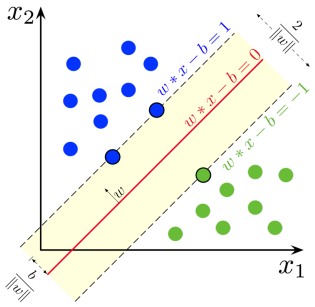
```
import numpy as np
import matplotlib.pyplot as plt
# Sigmoide
def sigmoide(z):
  return 1.0/(1 + np.exp(-z))

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()
```



1.5. Máquinas de soporte Vectorial

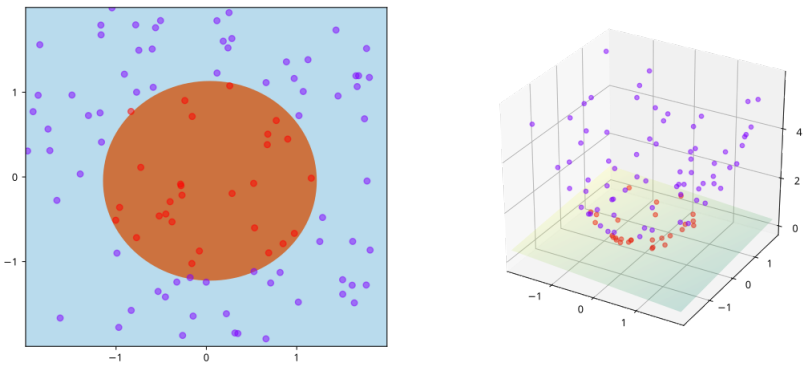
Una *Support Vector Machine* (SVM) es un algoritmo de aprendizaje automático que puede usarse tanto para regresión como para clasificación. Su objetivo es encontrar un hiperplano que separe los puntos de datos de una clase con respecto a la otra clase; pueden existir múltiples hiperplanos que separen las clases, el hiperplano óptimo es aquel que maximiza el *margen* entre dos clases.



https://bit.ly/4kVHyW6

El margen es la anchura máxima de la franja paralela al hiperplano que no contiene puntos en su interior. Las observaciones que marcan el límite del margen, es decir las más cercanas al hiperplano son los vectores de soporte. El margen definido como aquel que no contiene puntos suele llamarse margen *duro*, en ocasiones no es factible obtener este tipo de margen y se cambia por una margen *suave* que permite tener un número pequeño de puntos dentro del mismo.

Cuando el conjunto es separable linealmente, laa SVM encuentra los vectores que maximizan el margen. Sin embargo, para problemas más complejos es muy probable que no se encuentre separación lineal; en este caso, la obtención de los vectores de soporte se realiza con ayuda del llamado *truco del núcleo* (*kernel trick*) el cual consiste el transformar el conjunto de datos a un espacio de dimensión superior en la que se facilita la separación de clases con ayuda de una *función de núcleo*.



https://bit.ly/4kVHyW6

Existen diversas funciones de núcleo, las más comunes son función base radial (RBF o Gaussiana), polinomial y sigmoide.

Si bien las SVM se desarrollaron para tareas de clasificación (SVC), también pueden usarse para regresión (SVR); la idea básica es que la SVR identifica al hiperplano que mejor se ajusta a los datos con un margen de tolerancia especificado al mismo tiempo que minimiza el error de la predicción.

1.6. Funciones de predicción $k-$vecinos

Una función de predicción de este tipo utiliza un conjunto de observaciones en pares $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$ para los cuales se conocen todos los y_i valores objetivo; una observación objetivo es un par $(y_0, \mathbf{x}_0)$ para el que se desea calcular el valor de su variable objetivo y_0. La versión más sencilla de esta función de predicción para problemas de clasificación opera determinando las k observaciones más cercanas (similares) a $(y_0, \mathbf{x}_0)$, basada en las distancias entre $\mathbf{x}_0$ y $\mathbf{x}_i$; el valor de la predicción será la etiqueta más entre las observaciones más cercanas: sus $k-$vecinos. En el caso de variables cuantitativas, la predicción del algoritmo básico es la media de sus $k-$vecinos.

1.6.1. Notación

Una observación con valor objetivo y_0 desconocido, se denota como $\mathbf{z}_0 = (y_0, \mathbf{x}_0)$, en donde el vector predictor $\mathbf{x}_0$ de longitud p ha sido observado y se utilizará para predecir el valor de y_0. La función de predicción $f(\cdot|D)$ se construye a partir del conjunto de datos $D = \{\mathbf{z}_1, \ldots, \mathbf{z}_n\}$; la notación condicional de $f(\cdot|D)$ enfatiza el papel del conjunto de entrenamiento D. La predicción de y_0 se denota como $\hat{y}_0 = f(\mathbf{x}_0|D)$. Por ejemplo, si el objetivo es un valor **cuantitativo**, se puede utilizar una regresión lineal como función predictiva: $f(\mathbf{x}_0|D) = \mathbf{x}_0^T\hat{\boldsymbol{\beta}}$.

En general, la función de predicción no tiene una forma simple y se trata de un algoritmo de varios pasos que toma como entrada $\mathbf{x}_0$ y devuelve la predicción y_0. Si la variable es **cualitativa**, se asume que la variable objetivo es una etiqueta que identifica la pertenencia a algún grupo; si el número de grupos es g, entonces por convención el conjunto de etiquetas es $\{0, 1, \ldots, g-1\}$.

Además, es recomendable contar con funciones indicadoras que identifiquen la pertenencia a los grupos; el indicador de pertencia al grupo j se define como:

$$I_j(y) = \begin{cases} 1, & \text{si } y = j \\ 0, & \text{si } y \neq j \end{cases},$$

donde y es una etiqueta de grupo.

1.6.2. Métricas de distancia

Es común utilizar las distancias euclideana y Manhattan para calcular distancias entre vectores predictores.

La **distancia euclideana** entre los vectores $p-$dimensionales $\mathbf{x}_0$ y $\mathbf{x}_i$ se obtiene como:

$$d_E(\mathbf{x}_i, \mathbf{x}_0) = \left[\sum_{j=1}^{p}(x_{i,j} - x_{0,j})^2\right]^{1/2}$$

Si las variables del predictor difieren mucho con respecto a la variabilidad de las variables del conjunto de entrenamiento, es muy recomendable escalar cada variable con la desviación estándar de la misma; sin este proceso, las diferencias serán muy grandes y tenderán a sesgar la predicción. El escalamiento puede calcularse al momento de obtener la distancia como:

$$d_S(\mathbf{x}_i, \mathbf{x}_0) = \left[\sum_{j=1}^{p}\left(\frac{x_{i,j} - x_{0,j}}{s_j}\right)^2\right]^{1/2},$$

donde s_j es la desviación estándar estimada de la $j-$ésima variable predictora; la varianza se calcula como:

$$s_j^2 = (n-g)^{-1} \sum_{k=1}^{g} \sum_{i=1}^{n} I_k(y_i)(x_{i,j} - \bar{x}_{j,k})^2,$$

donde $\bar{x}_{j,k}$ es la media del atributo j dentro del grupo k; la presencia de $I_k(y_i)$ asegura que en la sumatoria interna sólo se consideren las observaciones del grupo correspondiente.

La **distancia Manhattan** (*city-block*) entre $\mathbf{x}_0$ y $\mathbf{x}_i$ se calcula como:

$$d_C(\mathbf{x}_i, \mathbf{x}_0) = \sum_{j=1}^{p} |x_{i,j} - x_{0,j}|,$$

de igual forma, puede utilizarse escalamiento para evitar sesgos en el cálculo.

En realidad en la mayoría de las aplicaciones de este algoritmo, es más impotante elegir el tamaño k del vecindario, dado que las métricas suelen devolver ordenamientos similares para los vecindarios.

Si un predictor consiste de p variables **cualitativas**, entonces la distancia más usada es la de **Hamming** para comparar $\mathbf{x}_0$ y $\mathbf{x}_i$; esta métrica determina el número de atributos que son diferentes entre ambos vectores; se calcula como:

$$d_H(\mathbf{x}_i, \mathbf{x}_0) = p - \sum_{j=1}^{p} I_{x_{i,j}}(x_{0,j}),$$

nótese que $I_{x_{i,j}}(x_{0,j})$ es 1 siempre que $x_{i,j} = x_{0,j}$ y 0 en otro caso; entonces la suma cuenta el número de atributos distintos entre ambos vectores.

1.6.3. La función de predicción de los $k-$vecinos más cercanos

La predicción de y_0 con el algoritmo de $k-$vecinos se obtiene determinando un vecindario de las k observaciones de entrenamiento más cercanas a $\mathbf{x}_0$; después se calcula la proporción de los k vecinos que pertenecen al grupo j y se predice el valor de y_0 como el grupo con la mayor proporción entre los vecinos. Esta regla de predicción es equivalente a predecir que $\mathbf{z}_0$ pertenece al grupo más común entre los k vecinos más próximos.

Formalmente, la función de predicción que estima la probabilidad de pertenencia al grupo j se define como la proporción de los miembros del grupo j entre los k vecinos más cercanos:

$$\widehat{Pr}(y_0 = j|\mathbf{x}_0) = \frac{n_j}{k}, j = 1, \ldots, g,$$

donde n_j es el número de vecinos dentro de los k más próximos que pertenecen a j. Resulta útil tener una expresión para $\hat{Pr}(y_0 = j|\mathbf{x}_0)$ en términos de variables indicadoras:

$$\widehat{Pr}(y_0 = j|\mathbf{x}_0) = k^{-1} \sum_{i=1}^{k} I_j\left(y_{[i]}\right)$$

Los corchetes denotan el valor objetivo del $i-$ésimo vecino más cercano; es decir, el vecino más próximo $\mathbf{z}_{[1]}$ tiene como valor objetivo $y_{[1]}$. Las probabilidades de pertenencia estimadas se agrupan como un vector:

$$\underset{g\times 1}{\widehat{\mathbf{p}}_0} = \left[\widehat{Pr}(y_0 = 1|\mathbf{x}_0), \ldots, \widehat{Pr}(y_0 = g|\mathbf{x}_0)\right]^T$$

El último paso para calcular la predicción obtiene el valor más grande dentro de $\widehat{\mathbf{p}}_0$:

$$f(\mathbf{x}_0|D) = \arg\max(\widehat{\mathbf{p}}_0)$$

La función arg máx devuelve el índice del elemento más grande de un vector, en este caso, de $\widehat{\mathbf{p}}_0$; en el caso de existir empate entre varios valores, devolverá el primero de ellos. Sin embargo, es mejor idea romper los empates de otra forma, por ejemplo incrementando el valor de k en uno y recalculando $\widehat{\mathbf{p}}_0$.

Un algoritmo computacional eficiente para la predicción con k vecinos consiste de dos funciones principales:

♢ obtener el arreglo ordenado $\mathbf{y}^o = \left(y_{[1]}, \ldots, y_{[n]}\right)$ para las entradas $\mathbf{x}_1, \ldots, \mathbf{x}_n$

♢ calcular los elementos de $\widehat{\mathbf{p}}$ y utilizar los primeros k elementos de $\mathbf{y}^o$

Las funciones de predicción $k-$vecinos más cercanos son conceptualmente simples y rivalizan con otras más sofisticadas en cuanto a la precisión de sus predicciones; el problema principal es que este tipo de algoritmos pueden ser computacionalmente muy costosos.

1.6.4. Ejemplo

1.7. Naive Bayes

1.8. Función de predicción: clasificador bayesiano multinomial

La función de predicción bayesiana (inocente, *naive*) es un algoritmo conceptual y computacionalmente simple; se dice que es *inocente* debido a que se presupone que las características son independientes una de la otra. En general su rendimiento no es el mejor cuando se trata de variables predictoras cuantitativas; se comporta aceptablemente bien cuando se tienen variables categóricas y es bastante bueno cuando se tienen variables categóricas con gran cantidad de categorías.

1.8.1. Introducción

Considerando problemas de predicción en los que las variables predictoras son categóricas; por ejemplo, los clientes que realizan sus compras en una tienda departamental se pueden clasificar en uno de múltiples grupos según sus hábitos de compra; los datos de entrenamiento pueden consistir de una enorme lista de productos adquiridos, cada uno puede ser identificado por una categoría, tales como perecederos, electrónica, *hardware*, etc. Fácilmente el número de grupos puede exceder los cientos, incluso miles; esto imposibilita el uso del algoritmo de $k-$vecinos; existen varios algoritmos alternativos para atacar este tipo de problemas, pero la *función de predicción multinomial bayesiana* (*inocente*) sobresale por la simplicidad de su algoritmo. Antes de los ejemplos de uso, se muestra el desarrollo de su fundamento matemático.

1.8.2. La función de predicción multinomial bayesiana

El problema se establece de la siguiente manera: la clase a la que pertenece la observación $\mathbf{z}_0 = (y_0, \mathbf{x}_0)$, es y_0 y $\mathbf{x}_0$ es su vector predictor; una función predictora $f(\cdot|D)$ se construye a partir de D, el conjunto de observaciones de entrenamiento. La predicción de y_0 es $\widehat{y}_0 = f(\mathbf{x}_0|D)$; la diferencia es que $\mathbf{x}_0$ es un vector de contadores: cada elemento de $\mathbf{x}_0$ almacena el número de veces en que un tipo particular (categoría) o nivel de una variable cualitativa fue observada.

Sea $x_{0,j}$ el número de veces en que un tipo t_j aparece en el *documento* F_0; por ejemplo, si un cliente compró tres productos en el departamento de electrónica, en su nota se puede contabilizar esa cantidad (la nota es el *documento* del que se extrae el dato). Entonces, $\mathbf{x}_0 = [x_{0,1}, \ldots, x_{0,n}]^T$ contiene las frecuencias de los tipos para el documento F_0 y existen n tipos diferentes dentro del conjunto de tipos T.

El *propietario* (clase) de F_0 se denota como y_0 y pertenece al conjunto de propietarios: $y_0 \in \{P_1, \ldots, P_m\}$; la probabilidad de que el tipo t_j se encuentre en F_0 está dada por:

$$\pi_{k,j} = \Pr\left(t_j \in F_0 | y_0 = P_k\right)$$

La probabilidad $\pi_{k,j}$ es específica para cada clase, pero no varía entre documentos del mismo propietario. Como t_j pertenece necesariamente a T, entonces $\sum_{j=1}^{n} \pi_{k,j} = 1$. La probabilidad de ocurrencia del tipo t_j puede variar entre diferentes propietarios; si las diferencias son substanciales, entonces se puede utilizar una función de predicción para discriminar entre los propietarios dado un vector de frecuencias $\mathbf{x}_0$.

Por ejemplo, si sólo existen dos tipos, dos propietarios y sus apariciones entre documentos son independientes, entonces la probabilidad de observar un vector particular, digamos $\mathbf{x}_0 = [27, 35]^T$, se puede calcular usando la distribución binomial:

$$\Pr\left(\mathbf{x}_0 | y_0 = P_k\right) = \frac{(27+35)!}{27! \times 35!} \pi_{k,1}^{27} \pi_{k,2}^{35}$$

Recordando que la variable aleatoria binomial calcula la probabilidad de observar x *éxitos* entre n *intentos*; la probabilidad de x éxitos y $n - x$ fallas se obtiene como:

$$\Pr\left(x\right) = \frac{n!}{x!\,(n-x)!} \pi^x \left(1-\pi\right)^{n-x},$$

para $x \in \{0, 1, \ldots, n\}$; la expresión anterior toma en cuenta el hecho que $\pi_2 = 1 - \pi_1$.

Los valores de cada $\pi_{k,i}$ son estimaciones basadas en conjuntos de datos previos y se sustituyen sus valores al momento del cálculo.

Con esto, la definición de la función de predicción es:

$$\widehat{y_0} = f\left(\mathbf{x}_0 | D\right) = \arg\max\left\{\Pr\left(\mathbf{x}_0 | y_0 = P_1\right), \ldots, \Pr\left(\mathbf{x}_0 | y_0 = P_m\right)\right\}$$

En general, el número de categorías es sustancialmente mayor que dos y el cálculo de la probabilidad requiere una extensión al de la distribución binomial. Al igual que en la distribución binomial, la versión multinomial supone que el número de ocurrencias de cierto tipo son eventos independientes, entonces la probabilidad de observar al vector de frecuencias $\mathbf{x}_0$ está dada por la función de probabilidad multinomial:

$$\Pr\left(\mathbf{x}_0 | y_0 = P_k\right) = \frac{\left(\sum_{i=1}^{n} x_{0,i}\right)!}{\prod_{i=1}^{n} x_{0,i}!} \pi_{k,1}^{x_{0,1}} \times \ldots \times \pi_{k,1}^{x_{0,n}}$$

El término que incluye factoriales se conoce como *coeficiente multinomial*, tal como el coeficiente binomial $n!/x!(n-x)!$. Al ser computacionalmente costoso, se busca evitar su cálculo y puede realizarse para determinar la función de predicción.

Tarea: _____

♢ Con $\mathbf{x}_0 = [27, 35]^T$, suponiendo que existen tres propietarios y que $\pi_{1,1} = 0.5$, $\pi_{1,2} = 0.3$, $\pi_{2,1} = 0.3$, $\pi_{2,2} = 0.4$, $\pi_{3,1} = 0.4$ y $\pi_{3,2} = 0.5$:

- Calcula $\Pr(\mathbf{x}_0|y_0 = P_k)$ para $k = 1, 2, 3$ y determina la predicción:

$$\widehat{y}_0 = \arg\max \{\Pr(\mathbf{x}_0|y_0 = P_1), \Pr(\mathbf{x}_0|y_0 = P_2), \Pr(\mathbf{x}_0|y_0 = P_3)\}$$

_____*

1.8.2.1. Probabilidad posterior La función predictora de Bayes puede mejorarse al tomar en cuenta información previa. Por ejemplo, al tratar de asignar un documento a algún propietario se puede tomar en cuenta cierto conocimiento histórico sobre documentos ya clasificados para obtener el valor de las *probabilidades previas*. Incluso en ausencia de más evidencias (p.e. el vector de frecuencias), al buscar determinar al propietario de algún documento, se debería elegir a aquel que tenga la mayor cantidad hasta el momento.

La fórmula de Bayes conjunta la información contenida en el vector de frecuencias y la información previa para determinar la probabilidad posterior de la propiedad de algún documento:

$$\Pr(y_0 = P_k|\mathbf{x}_0) = \frac{\Pr(\mathbf{x}_0|y_0 = P_k) \times \Pr(y_0 = P_k)}{\Pr(\mathbf{x}_0)}$$

La predicción de la propiedad será aquella con la mayor probabilidad subsecuente; la predicción se obtiene como:

$$\begin{aligned} \widehat{y}_0 &= \arg\max_k \{\Pr(y_0 = P_k|\mathbf{x}_0)\} \\ &= \arg\max_k \left\{ \frac{\Pr(\mathbf{x}_0|y_0 = P_k) \times \Pr(y_0 = P_k)}{\Pr(\mathbf{x}_0)} \right\} \end{aligned}$$

El denominador $\Pr(\mathbf{x}_0)$ escala las probabilidades subsecuentes por el mismo factor y no afecta el orden de las mismas; por tanto puede simplificarse a una expresión más práctica:

$$\widehat{y}_0 = \arg\max_k \{\Pr(\mathbf{x}_0|y_0 = P_k) \times \Pr(y_0 = P_k)\}$$

Como el coeficiente multinomial depende solamente del vector $\mathbf{x}_0$ y tiene el mismo valor para cualquier P_k, puede ser ignorado en el cálculo de $\Pr(\mathbf{x}_0|y_0 = P_k)$, dado que sólo nos interesa determinar cual de las probabilidades posteriores es mayor.

Las probabilidades multinomiales ($\pi_{k,j}$) son desconocidas, pero pueden estimarse a partir de la información previa, evitando el cálculo del coeficiente multinomial; así, la función de Bayes en términos de las probabilidades estimadas es:

$$\begin{aligned} \widehat{y}_0 &= \arg\max_k \left\{ \widehat{\Pr}(y_0 = P_k|\mathbf{x}_0) \right\} \\ &= \arg\max_k \left\{ \widehat{\pi}_{k,1}^{x_{0,1}} \times \ldots \times \widehat{\pi}_{k,n}^{x_{0,n}} \times \widehat{\pi}_k \right\} \end{aligned},$$

donde $x_{0,j}$ es la frecuencia de ocurrencia del tipo t_j en el documento F_0; la probabilidad estimada $\widehat{\pi}_{k,j}$ es la frecuencia relativa de la ocurrencia de t_j entre todos los documentos pertenecientes a P_k; finalmente, $\widehat{\pi}_k = \widehat{\Pr}(y_0 = P_k)$ es la probabilidad previa estimada de que el documento pertenezca a P_k.

Una vez que se observa al vector $\mathbf{x}_0$, se utiliza la información que contiene para actualizar las probabilidades posteriores. Si no se cuenta con información previa, se deben utilizar probabilidades previas *no informativas*: $\pi_1 = \ldots = \pi_g = 1/g$, suponiendo que hay g propietarios.

Para utilizar la fórmula de Bayes, se necesita estimaciones de las probabilidades $\pi_{k,j}$, $j = 1, \ldots, n$ y $k = 1, \ldots, g$; para obtenerlas se usan las proporciones muestrales o probabilidades empíricas:

$$\widehat{\pi}_{k,j} = \frac{x_{k,j}}{n_k},$$

donde $x_{k,j}$ es el número de ocurrencias de t_j en los documentos pertenecientes a P_k y $n_k = \sum_j x_{k,j}$ es el total de todas la frecuencias de tipos para el propietario P_k. Se puede presentar un subdesbordamiento (*underflow*) si los exponentes de $\widehat{\pi}_{k,j}^{x_{0,j}}$ son grandes y las bases son pequeñas; para evitarlo se busca al propietario con la mayor probabilidad *logarítmica-posterior*:

$$\arg\max_k \left\{ \widehat{\Pr}(y_0 = P_k|\mathbf{x}_0) \right\} = \arg\max_k \left\{ \log \left[\widehat{\Pr}(y_0 = P_k|\mathbf{x}_0) \right] \right\}$$

Con esto, la versión final de la función de predicción bayesiana multinomial es:

$$\begin{aligned} \widehat{y}_0 &= \arg\max_k \left\{ \log \left[\widehat{\Pr}(y_0 = P_k|\mathbf{x}_0) \right] \right\} \\ &= \arg\max_k \left\{ \log(\widehat{\pi}_k) + \textstyle\sum_{j=1}^{n} x_{0,j} \log(\widehat{\pi}_{k,j}) \right\} \end{aligned}$$

Tarea:

♢ Para el mismo ejemplo, suponiendo las probabilidades previas $\pi_1 = 0.8$, $\pi_2 = \pi_3 = 0.1$, determina las probabilidades posteriores $\Pr(y_0 = P_k|\mathbf{x}_0)$ para $k = 1, 2, 3$ y determina la predicción $\widehat{y}_0$.

*****

1.8.3. Ejemplo

Construcción de un detector de *spam* en textos cortos.

1.9. XGBoost

1.9.1. Árboles de decisión/regresión

Los árboles de decisión son un tipo importante de modelos predictivos de aprendizaje artificial (*machine learning*). Existen varios algoritmos para este tipo de árboles desde hace varias décadas y algunas variantes como *random forest* se consideran como técnicas muy poderosas en el área de minería de datos (*data mining*).

El término *Classification and Regression Trees* (CART) fue introducido por Leo Breiman (https://bit.ly/2RvIgwl) en 1984 para referirse a árboles de decisión y regresión que pueden usarse como modelos de clasificación o regresión. El algoritmo CART es el fundamento de otros importantes algoritmos tales como *bagged decision trees*, *random forest* y *boosted decision trees* (https://bit.ly/2NtIUXZ). La idea básica es dividir repetidamente los registros disponibles buscando maximizar la *homogeneidad* en los conjuntos obtenidos en dicha división.

Es un modelo jerárquico de aprendizaje que puede ser usado tanto para clasificación como para regresión: permiten explorar relaciones complejas entre entradas y salidas sin necesidad de hacer suposiciones sobre los datos. Los árboles de decisión pueden verse como una función f que realiza una estimación de un *mapeo* desde un espacio de entrada X hacia un espacio objetivo y: $y = f(X)$. Para el caso de la clasificación, el espacio objetivo y es numérico: $y \in \mathbb{R}$; para la regresión, los valores de y son categóricos: $y \in \{C_1, C_2, \ldots\}$. Se trata de un modelo de aprendizaje supervisado; por tanto, se tienen muestras con la salida esperada. La representación más común para CART es un árbol binario.

Un árbol de decisión divide el espacio X en regiones locales utilizando alguna medida de distancia y el objetivo es determinar particiones bien separadas y homogéneas. Las regiones se dividen de acuerdo a preguntas de prueba y, con esto, se obtiene una división $n-$aria, de forma que mientras se recorre el árbol, en cada nodo se realiza toman decisiones.

Ejemplo 1

La figura 4 se muestra un ejemplo sencillo. Los datos se encuentran en dos dimensiones $\{x_1, x_2\}$, también llamadas características, variables ó atributos. Se trata de una clasificación binaria con clases *Redonda* y *Cuadrada*. En el nodo raíz se pregunta si $x_1 > w_{10}$ que divide los datos en un nodo hijo (derecho) con instancias de la clase R y un nodo hijo (izquierdo) que tiene instancias tanto de R como de C. En el hijo izquierdo se realiza una segunda pregunta $x_2 > w_{20}$ para obtener finalmente instancias separadas de las clases R y C. Es importante notar que los resultados obtenidos realizan particiones disjuntas en las variables de entrada.

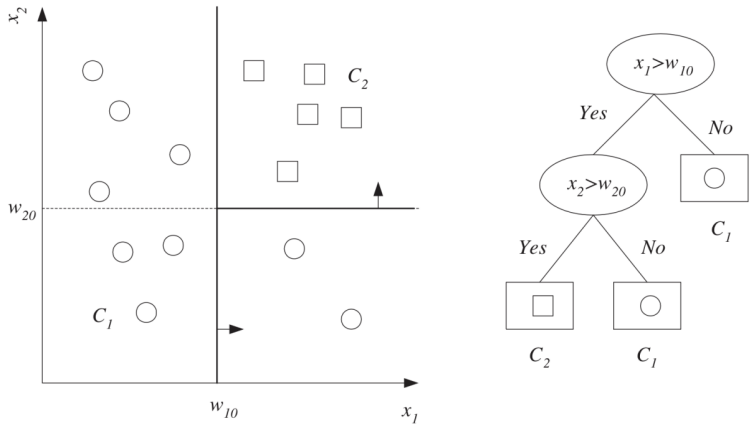


Figure 4: Ejemplo de árbol de decisión/regresión (derecha) y sus datos (izquierda)

Ejemplo 2

La figura 5 muestra otro ejemplo sencillo para determinar si se debe cargar paraguas o no, de acuerdo a la predicción del estado del tiempo.

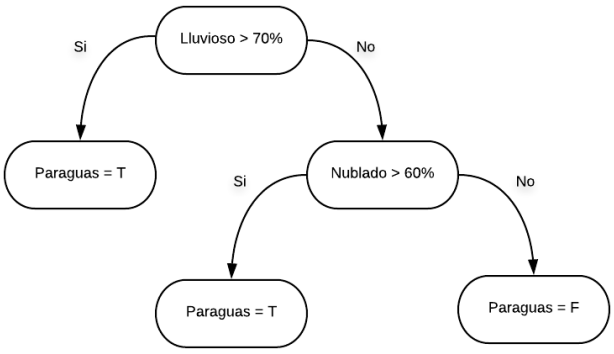


Figure 5: Ejemplo de árbol de decisión para llevar paraguas

El algoritmo 2 muestra la forma de obtener los valores para las particiones.

Algorithm 2 Obtención de particiones

elegir una variable x_i

- ♢ elegir algún valor s_i para x_i que divida los datos de entrenamiento en dos particiones (no necesariamente iguales)
- ♢ medir la *pureza* (homogeneidad) resultante en cada partición
- ♢ usar otro valor s_j para x_i buscando incrementar la pureza de las particiones hasta alcanzar el umbral de *pureza aceptable*

repetir el proceso de partición con una variable diferente (posiblemente alguna usada previamente)
cada valor obtenido se convierte en un nodo en el árbol

Para determinar la homogeneidad de las particiones, se tienen dos posibilidades:

1.9.1.1. Grado de impureza de Gini El índice de impureza en un rectángulo A que contiene m clases, se calcula como:

$$I(A) = 1 - \sum_{i=1}^{m} p_i^2$$

Con p la proporción de casos en el rectángulo A que pertenecen a la clase i. Es importante notar que:

- ♢ $I(A) = 0$ cuando todos los casos pertenecen a la clase i; es decir, es totalmente homogénea.
- ♢ El valor máximo sucede cuando todas las clases se encuentran igualmente representadas (0.5 para el caso binario).

1.9.1.2. Grado de entropía El grado de entropía en un área A que contiene m clases, se calcula como:

$$E(A) = \sum_{i=1}^{m} p_i \times log_2(p_i)$$

Con p la proporción de casos en A que pertenecen a la clase i. Es importante notar:

♢ $E(A) = 0$ cuando todos los casos pertenecen a la clase i; es decir, es totalmente homogénea.

♢ El valor máximo $log_2(m)$ sucede cuando todas las clases se encuentran igualmente representadas.

Problemas con CART

♢ Pueden ser poco robustos: un cambio pequeño en los datos de entrenamiento generan grandes cambios en la estructura del árbol.

♢ Obtener el mejor árbol de decisión es $NP - completo$; por tanto, en la práctica para su construcción se utilizan heurísticas basadas en óptimos locales.

♢ No es tan difícil obtener modelos sobreajustados que no generalizan bien las muestras; por esto es necesario establacer un límite en la obtención de particiones.

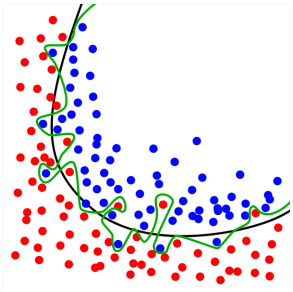


Figure 6: Ejemplo de sobreajuste

Ejemplo 3

Para el caso de variables categóricas el proceso es más simple. En la figura 7 se observa un ejemplo de datos categóricos para determinar si se debe jugar tennis o no, de acuerdo a la predicción del estado del tiempo.

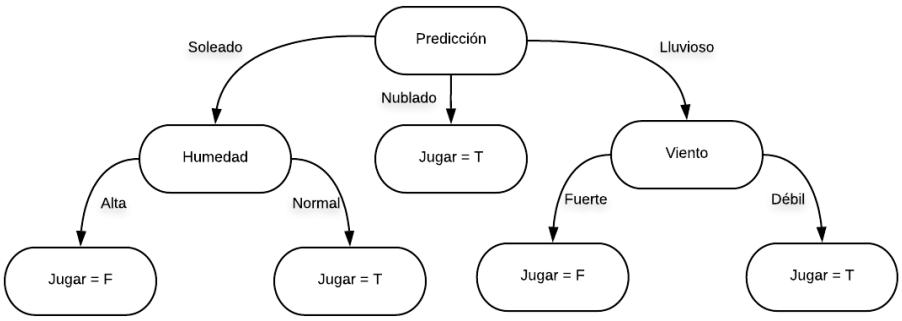


Figure 7: Ejemplo de árbol de decisión con variables categóricas

Extracción de reglas

Una vez constuidos, los árboles de decisión son altamente interpretables y es fácil entender la información que representan como reglas del tipo *if - then*: cada camino desde un nodo hasta una hoja es una conjunción de condiciones que deben ser satisfechas para llegar a ese nodo terminal. Para el último ejemplo, tenemos:

♢ **if** P=S **and** H=N **then** J=T

♢ **if** P=N **then** J=T

♢ **if** P=Ll **and** V=D **then** J=T

1.9.2. Función de predicción: *eXtreme Gradient Boosting* (*XGBoost*)

XGBoost es una biblioteca optimizada y distribuída de mejora del gradiente diseñada para ser altamente eficiente, flexible y portable. implementa algoritmos de aprendizaje automático bajo la idea del mejora del gradiente (*Gradient Boosting*). *XGBoost* provee mejora en la construcción de árboles paralelizada.

XGBoost es una de las implementacinoes más populares y eficientes del algoritmo *Gradient Boosted Trees*, un método supervisado de aprendizaje basado en aproximación de funciones al optimizar las funciones de pérdida mientras aplica varias técnicas de regularización.

La propuesta original se encuentra en: https://arxiv.org/pdf/1603.02754.pdf.

1.9.2.1. Ejemplo

Predictor de quema de calorías con ejercicio.

Tarea: _____

♢ Aplicar el modelo *XGBoost* al conjunto de datos https://www.kaggle.com/c/digit-recognizer/, aquí hay un tutorial con KNN *desde cero*: https://www.kaggle.com/code/fergusmclellan/digit-recognition-using-python-knn-from-scratch

♢ Comparar los resultados de *k-vecinos*, *bayesiano multinmial* y *XGBoost*