Taller de Verano: 100 páginas de Machine Learning

Maikol Solís

2024-02-12

Tabla de contenidos

1 Taller de verano

2 Cómo funciona el aprendizaje supervisado

Veremos el caso de las máquinas de soporte vectorial (SVM) para clasificación.

- **Paso #1: Cargar librerías**

```
import matplotlib.pyplot as plt

from sklearn import svm
from sklearn.datasets import make_blobs
from sklearn.inspection import DecisionBoundaryDisplay

from scipy.stats import distributions
from numpy import sum
import numpy as np
```

- **Paso #2: Crear datos**

Se crean 40 puntos usando la función `make_blobs`. Esta crea un conjunto de puntos separados en dos grupos.

```
X, y = make_blobs(n_samples=40, centers=2, random_state=6)
```

- **Paso #3: Crear el modelo**

```
clf = svm.SVC(kernel="linear", C=1000)
```

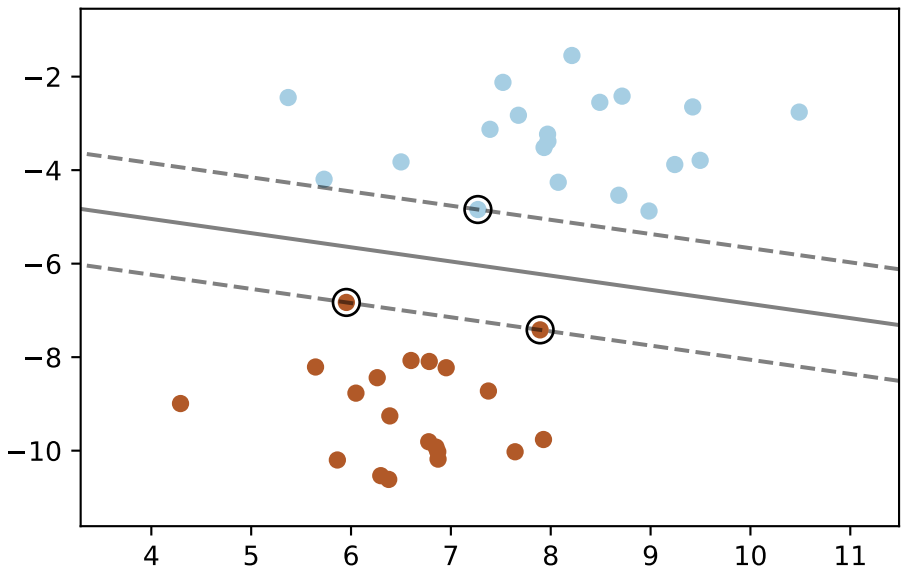
- **Paso #4: Entrenar el modelo**

```
clf.fit(X, y)
```

```
SVC(C=1000, kernel='linear')
```

- **Paso #5: Visualizar el modelo**

```
plt.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=plt.cm.Paired)

# plot the decision function
ax = plt.gca()
DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    plot_method="contour",
    colors="k",
    levels=[-1, 0, 1],
    alpha=0.5,
    linestyles=["--", "-", "--"],
    ax=ax,
)
# plot support vectors
ax.scatter(
    clf.support_vectors_[:, 0],
    clf.support_vectors_[:, 1],
    s=100,
    linewidth=1,
    facecolors="none",
    edgecolors="k",
)
plt.show()
```

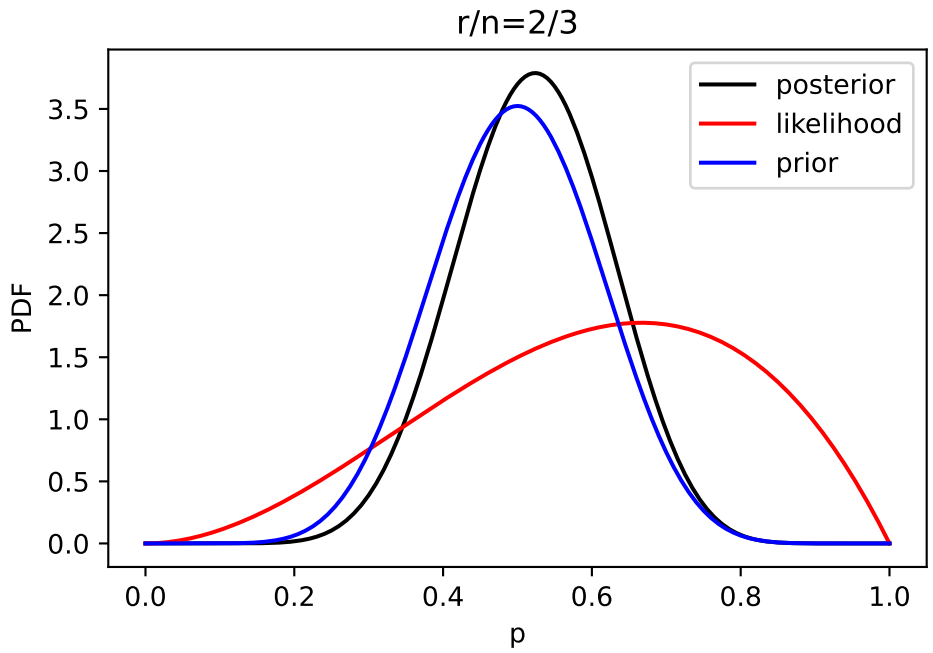


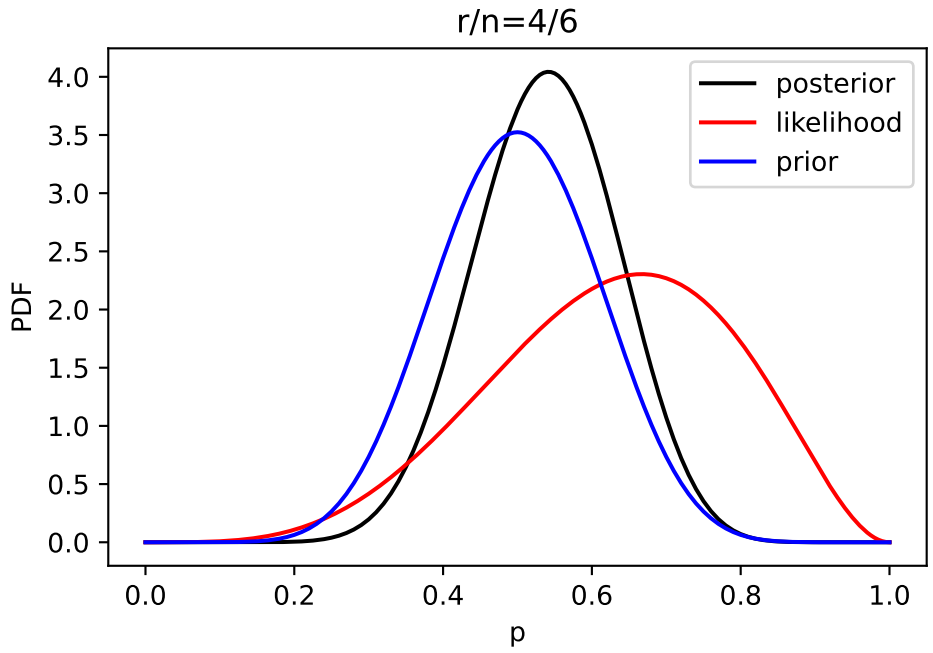
- **Referencias**

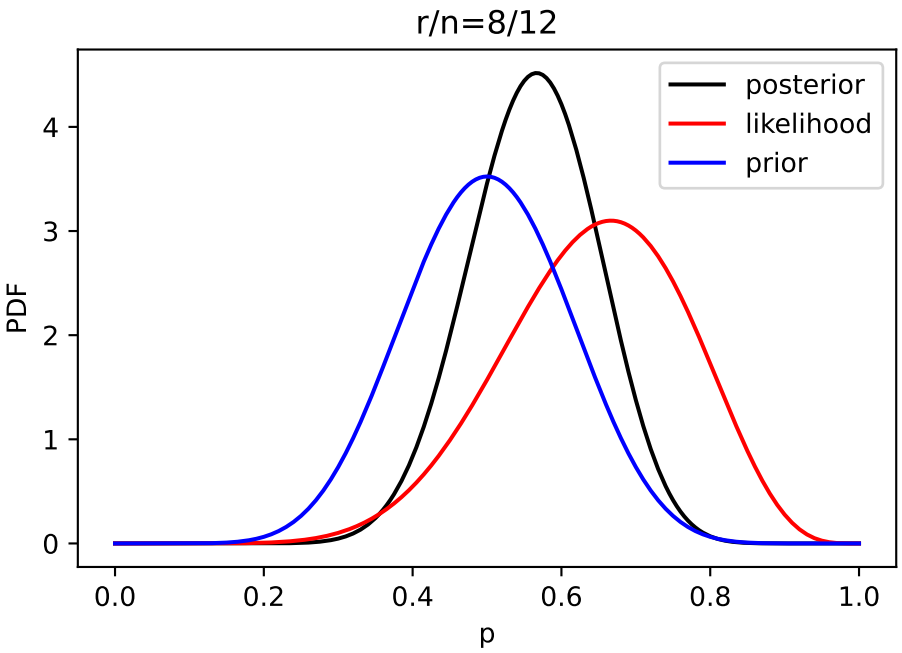
1. https://scikit-learn.org/stable/modules/svm.html#
2. https://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html#sphx-glr-auto-examples-svm-plot-separating-hyperplane-py

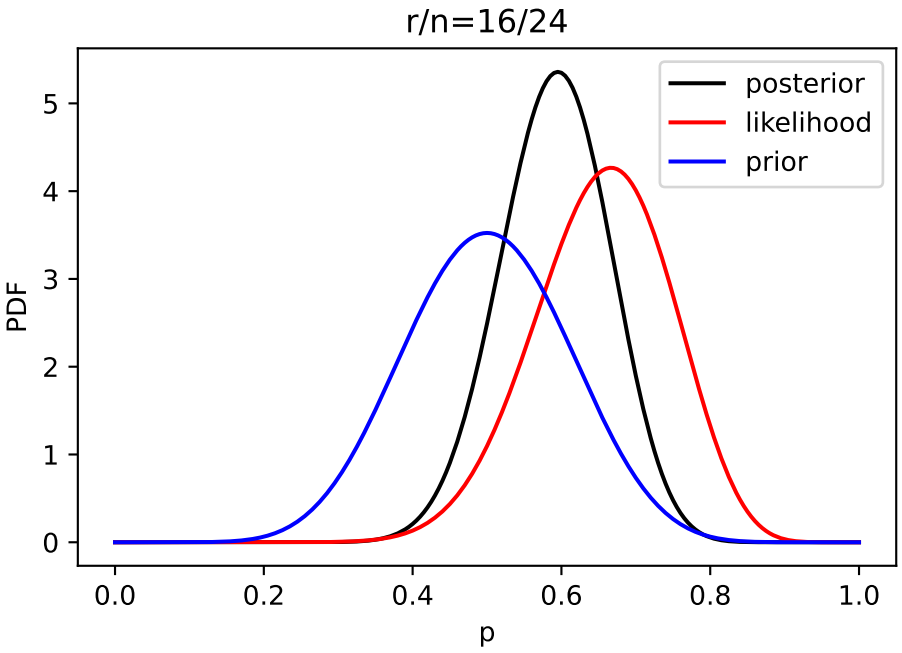
3 Estimación de parametros bayesiano

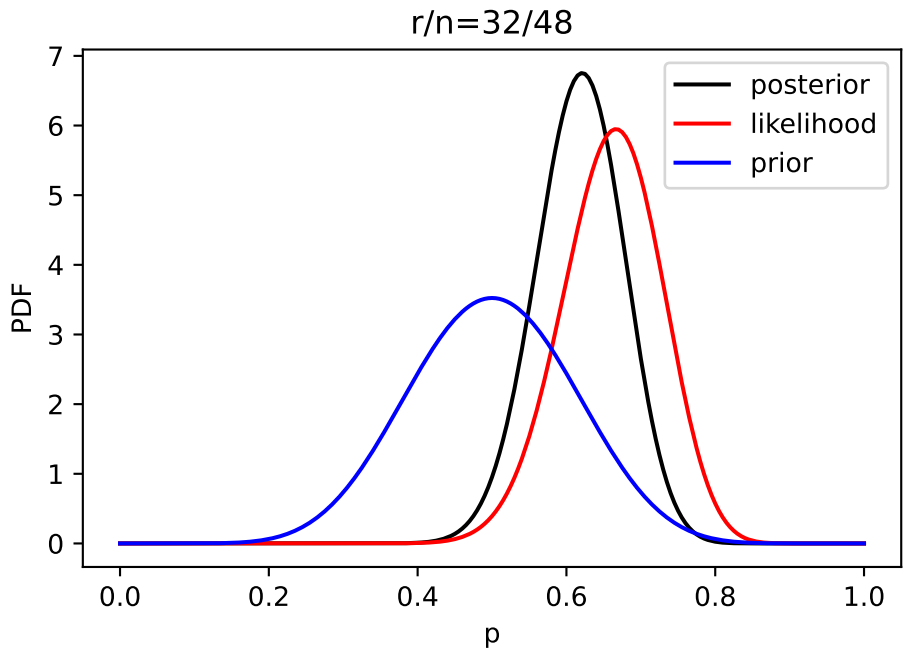
```
alpha = 10
beta = 10
n = 20
Nsamp = 201  # no of points to sample at
p = np.linspace(0, 1, Nsamp)
deltap = 1./(Nsamp-1)  # step size between samples of p

prior = distributions.beta.pdf(p, alpha, beta)

for i in range(1, 9):

    r = 2**i
    n = (3.0/2.0)*r
    like = distributions.binom.pmf(r, n, p)
    like = like/(deltap*sum(like))  # for plotting convenience only
    post = distributions.beta.pdf(p, alpha+r, beta+n-r)

    # make the figure
    plt.figure()
    plt.plot(p, post, 'k', label='posterior')
    plt.plot(p, like, 'r', label='likelihood')
    plt.plot(p, prior, 'b', label='prior')
    plt.xlabel('p')
    plt.ylabel('PDF')
    plt.legend(loc='best')
    plt.title('r/n={}/{:.0f}'.format(r, n))
    plt.show()
```

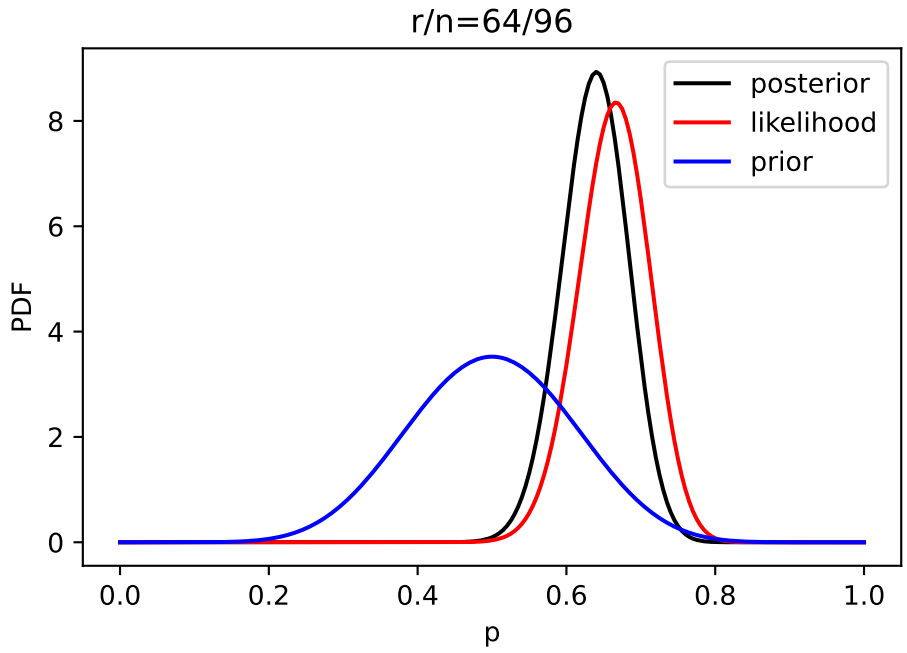

r/n=2/3
posterior
likelihood
prior
PDF
p
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.0
0.2
0.4
0.6
0.8
1.0

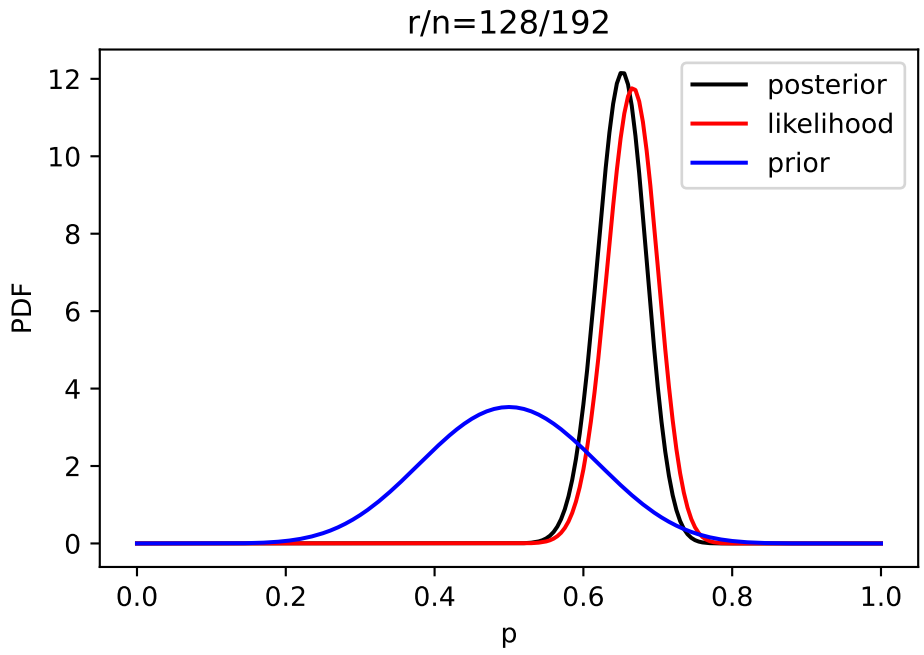

r/n=4/6
posterior
likelihood
prior
PDF
p
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0.0
0.2
0.4
0.6
0.8
1.0

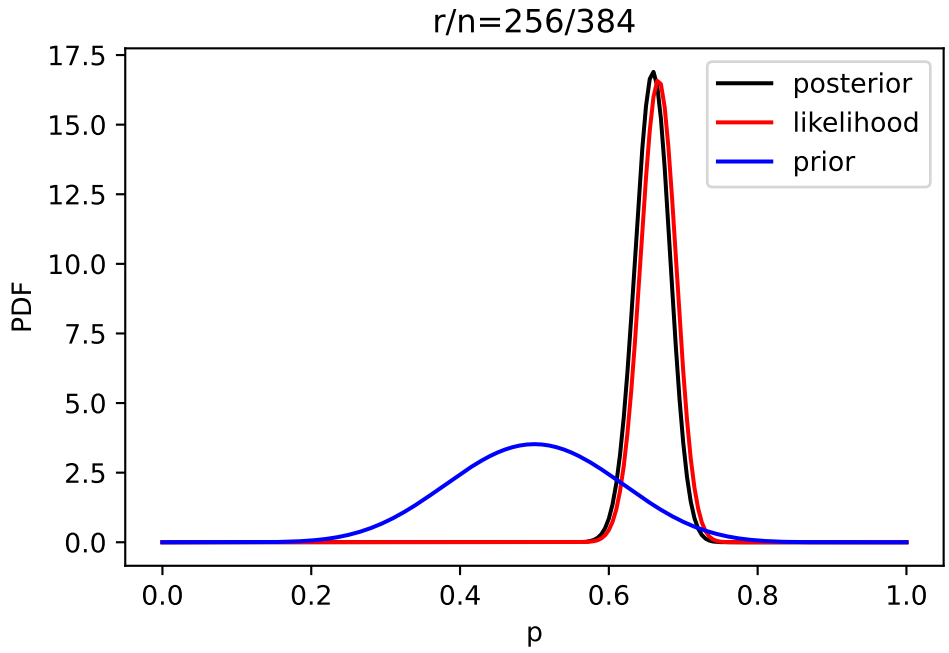

r/n=8/12
posterior
likelihood
prior
PDF
p
0
1
2
3
4
0.0
0.2
0.4
0.6
0.8
1.0


r/n=16/24
posterior
likelihood
prior
PDF
p
0
1
2
3
4
5
0.0
0.2
0.4
0.6
0.8
1.0


r/n=32/48
posterior
likelihood
prior
PDF
p
0
1
2
3
4
5
6
7
0.0
0.2
0.4
0.6
0.8
1.0


r/n=64/96
posterior
likelihood
prior
PDF
p
0
2
4
6
8
0.0
0.2
0.4
0.6
0.8
1.0


r/n=128/192
posterior
likelihood
prior
PDF
p
0
2
4
6
8
10
12
0.0
0.2
0.4
0.6
0.8
1.0


r/n=256/384
posterior
likelihood
prior
PDF
p
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0.0
0.2
0.4
0.6
0.8
1.0

4 Día 2

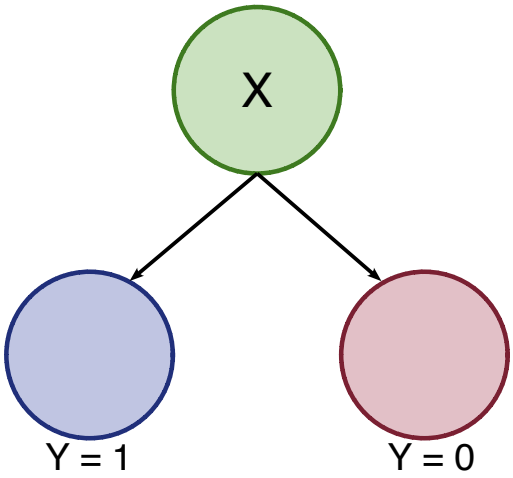
4.1 Regresión Lineal

4.2 Regresión Logística

4.3 Decision Trees

4.3.1 Definición

Un árbol de decisión (DT) es un grafo no cíclico que se utiliza para tomar decisiones (clasificar). En cada nodo (rama) del grafo se evalúa uno de los *features*. Si el resultado de la evaluación es cierto (o está debajo de un umbral), se sigue la rama de la izquierda, si no se va a la derecha.



Por lo tanto, los DT son un modelo no paramétrico.

Para crear el DT, se **intenta** optimizar el promedio de la máxima verosimilitud:

$$\frac{1}{N}\sum_{i=1}^{N}\left(y_i \ln f_{ID3}(x_i) + (1-y_i)\ln\left(1-f_{ID3}(x_i)\right)\right)$$

donde f_{ID3} es un DT y $f_{ID3}(x) \stackrel{\text{def}}{=} Pr(y=1|x)$

4.3.2 Construcción

Para construir el árbol, en cada nodo de decisión, se intenta minimizar la entropía de la información.

La entropía de un conjunto $\mathcal{S}$ viene dada por:

$$H(S) \stackrel{\text{def}}{=} -f^S_{ID3} \log_2(f^S_{ID3}) - (1 - f^S_{ID3}) \log_2(1 - f^S_{ID3})$$

Y si un grupo se divide en dos, la entropía es la suma ponderada de cada subconjunto:

$$H(S_-, S_+) \stackrel{\text{def}}{=} \frac{|S_-|}{|S|} H(S_-) + \frac{|S_+|}{|S|} H(S_+)$$

4.3.3 Ejemplo

Consideremos los siguientes datos:

Atributos:

- Edad: viejo (v), media-vida(m), nuevo (nv)
- Competencia: no(n), sí(s)
- Tipo: software (swr), hardware (hwr)

Edad	Competencia	Tipo	Ganancia
v	s	swr	baja
v	n	swr	baja
v	n	hwr	baja
m	s	swr	baja
m	s	hwr	baja
m	n	hwr	sube
m	n	swr	sube
nv	s	swr	sube
nv	n	hwr	sube
nv	n	swr	sube

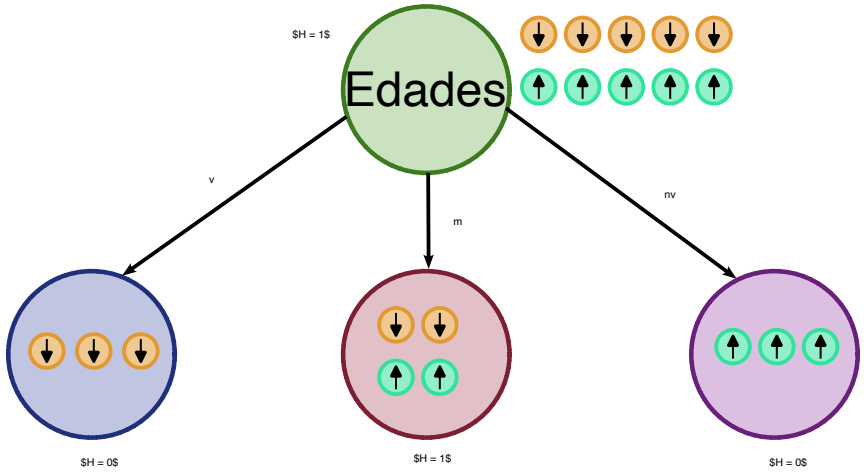
Cálculo de las entropías: Primero se tiene que probar todos los features para ver cuál tiene mayor ganancia de información (reduce la entropía)

Entropía total:

$$H(S) = \text{Entropía de los casos baja} + \text{Entropía de los casos sube}$$

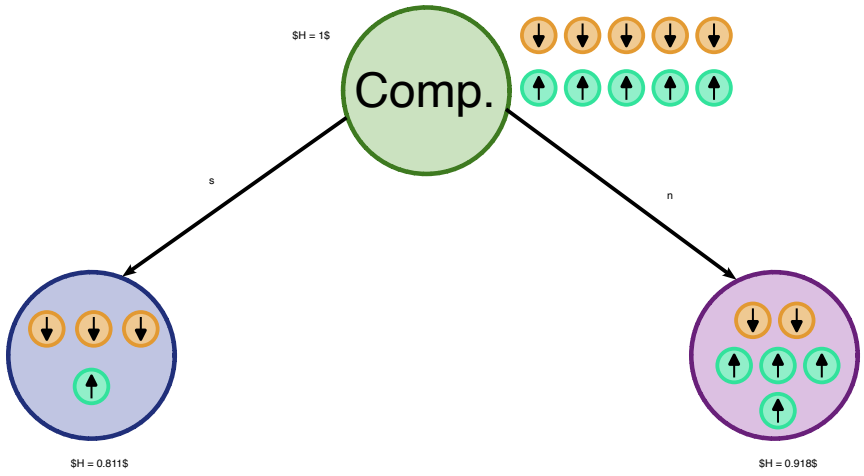
$$H(s) = -\frac{5}{10} * \log_2(\frac{5}{10}) - \frac{5}{10} * \log_2(\frac{5}{10}) = 1$$

Ahora vamos a decidir la primera separación con las edades



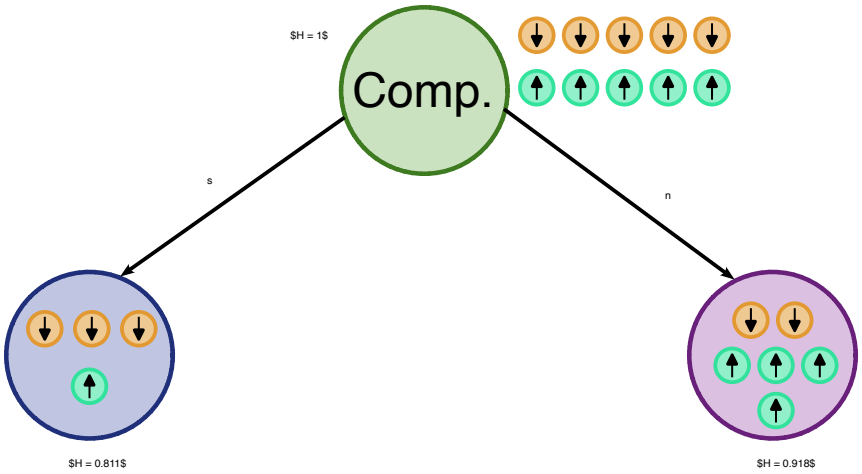
$H = \frac{3}{10} \cdot 0 + \frac{4}{10} \cdot 1 + \frac{3}{10} \cdot 0 = 0.4$

Ahora vamos a decidir la primera separación con la competencia



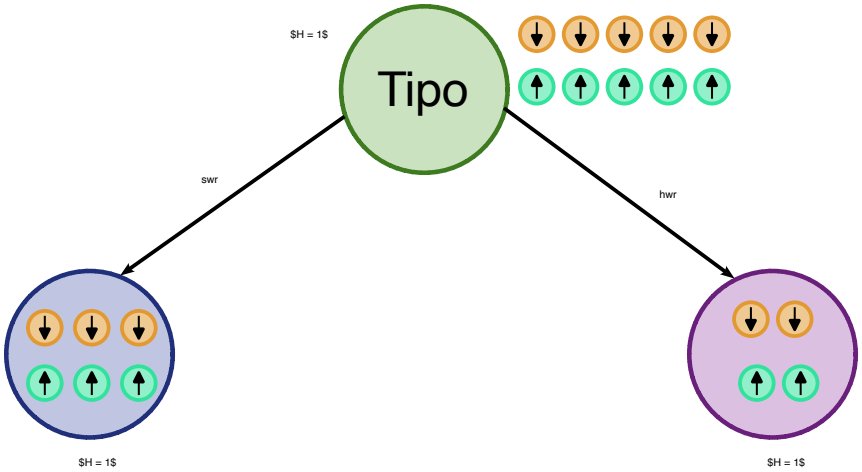
$H = \frac{4}{10} \cdot 0.811 + \frac{6}{10} \cdot 0.918 = 0.8752$

Ahora vamos a decidir la primera separación con las edades



$H = \frac{4}{10} \cdot 0.811 + \frac{6}{10} \cdot 0.918 = 0.8752$

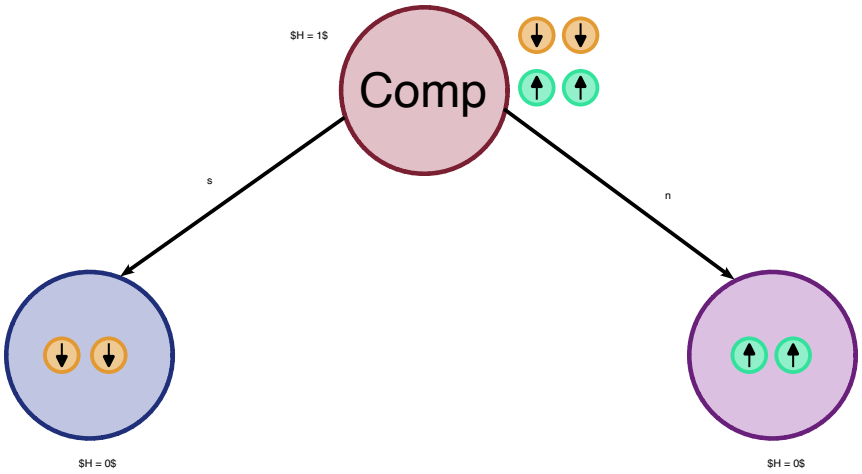
Ahora vamos a decidir la primera separación con el tipo



$H = \frac{6}{10} \cdot 1 + \frac{4}{10} \cdot 1 = 1$

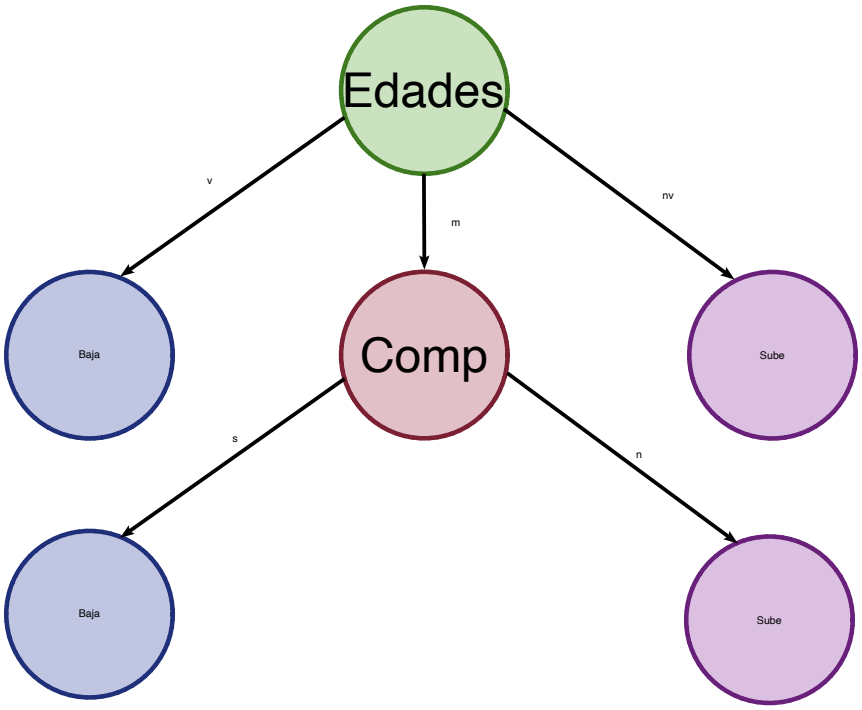
Concluimos que lo que nos da la máxima ganancia de información es primero decidir por edades, eso nos deja dos nodos hoja y un nodo rama que debemos volver a separar.

Ahora vamos a buscar el segundo nivel, donde vamos a separar el grupo que tiene edades medias por competencia:

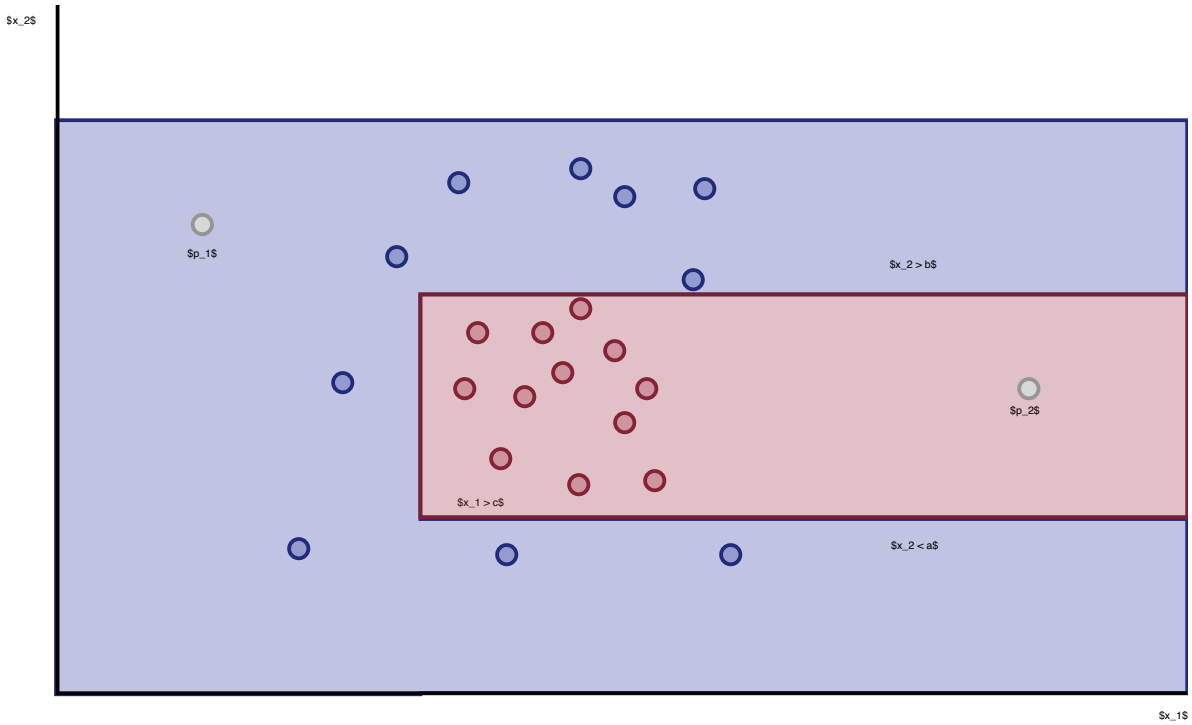


$H = \frac{2}{4} \cdot 0 + \frac{2}{4} \cdot 0 = 0$

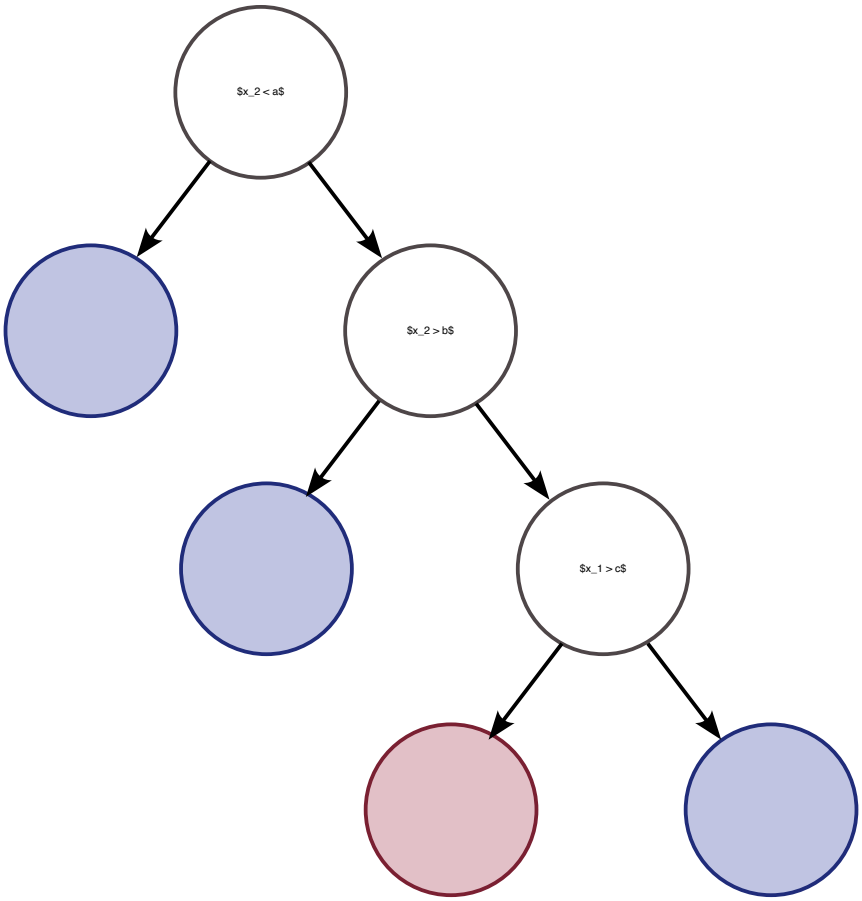
Con esto ya se clasificaron todos los datos, puesto que terminamos solo con nodos hojas:



Esto también se puede hacer con valores numéricos, que de hecho, es lo que se puede hacer con scikit learn



y con esto se obtiene este árbol de decisión:



4.3.4 Comandos básicos en python

Estos son los comandos básicos en python

```
#| label: dibujoArbol01
#| fig-cap: "Árbol de decisión"
from sklearn import tree
X = # Lista con los features (lista de listas)
Y = # Lista con los labels
# Se define la variable que tendrá el árbol
clf = tree.DecisionTreeClassifier()
# Se calcula el árbol
clf = clf.fit(X, Y)
# Se utiliza el árbol para predecir el label de un dato nuevo
clf.predict_proba(X0)
```

```
# Se dibuja el árbol
tree.plot_tree(clf)
```

y este sería un ejemplo sencillo en python:

Primero creamos los datos

```
from sklearn import tree
from sklearn.datasets import make_blobs
from sklearn.inspection import DecisionBoundaryDisplay
import matplotlib.pyplot as plt
import numpy as np

# Creación de los datos
X, Y = make_blobs(n_samples=200, centers=4, random_state=6)
plt.scatter(X[:, 0], X[:, 1], c=Y, s=30)
plt.title("Datos originales")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

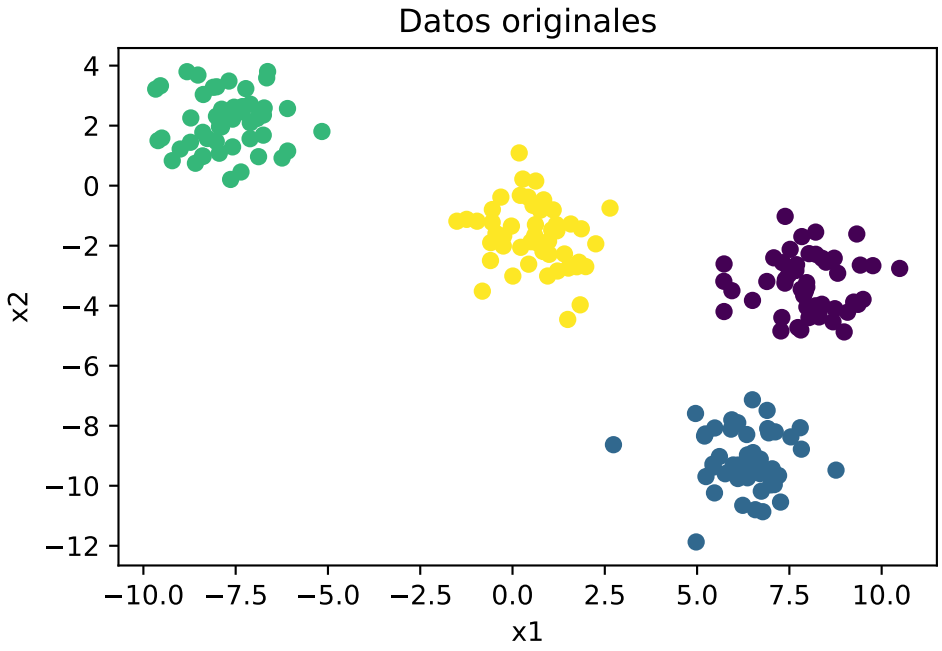


Figura 4.1: Ejemplo hecho en python: datos

Luego se crea el arbol

```
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, Y)
tree.plot_tree(clf)
plt.show()
```

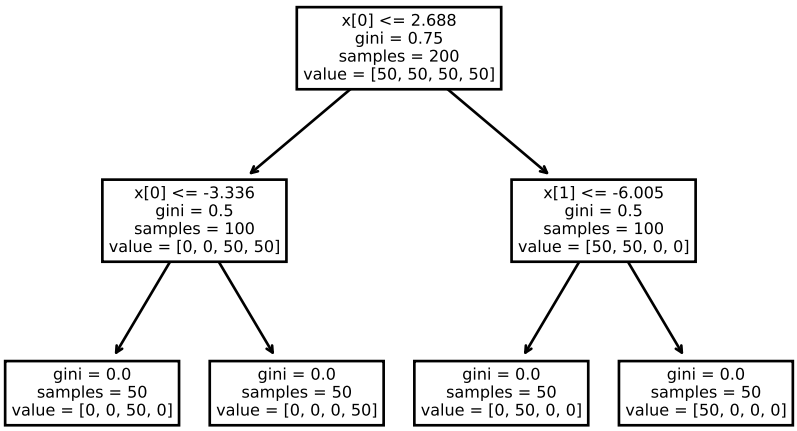


Figura 4.2: Ejemplo hecho en python: arbol

y por último, dibujamos las separaciones

```
DecisionBoundaryDisplay.from_estimator(clf, X, response_method="predict")
plt.scatter(X[:, 0], X[:, 1], c=Y, s=30)
plt.show()
```

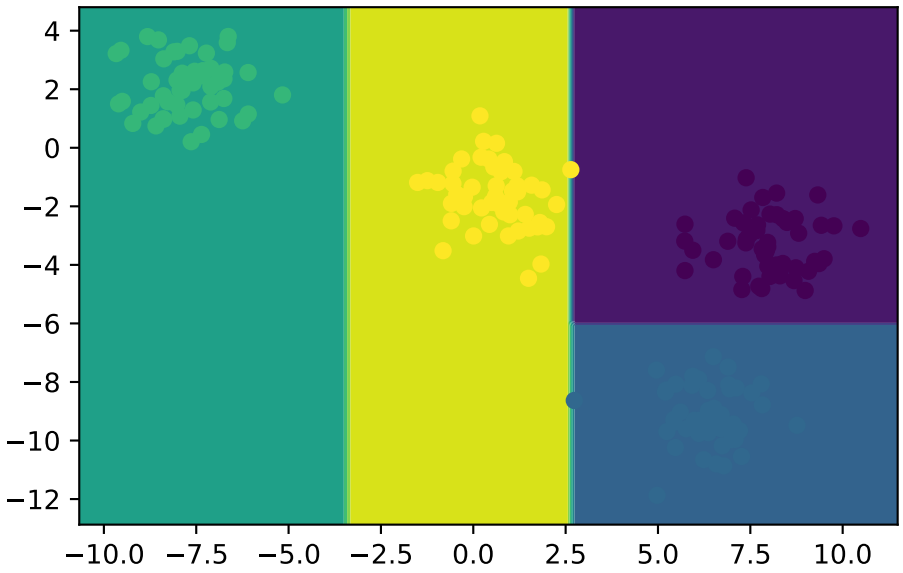


Figura 4.3: Ejemplo hecho en python: separación

y con esto se puede aplicar el árbol

```
print(clf.predict([[5.0, 1.0]]))
print(clf.predict([[-2.0, -1.0]]))
print(clf.predict([[6.0, -6.0]]))
```

```
[0]
[3]
[0]
```

y lo que devuelve es el número de grupo al que pertene el dato

4.4 K-Nearest Neighbors (KNN)

4.4.1 Carga de paquetes

```
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

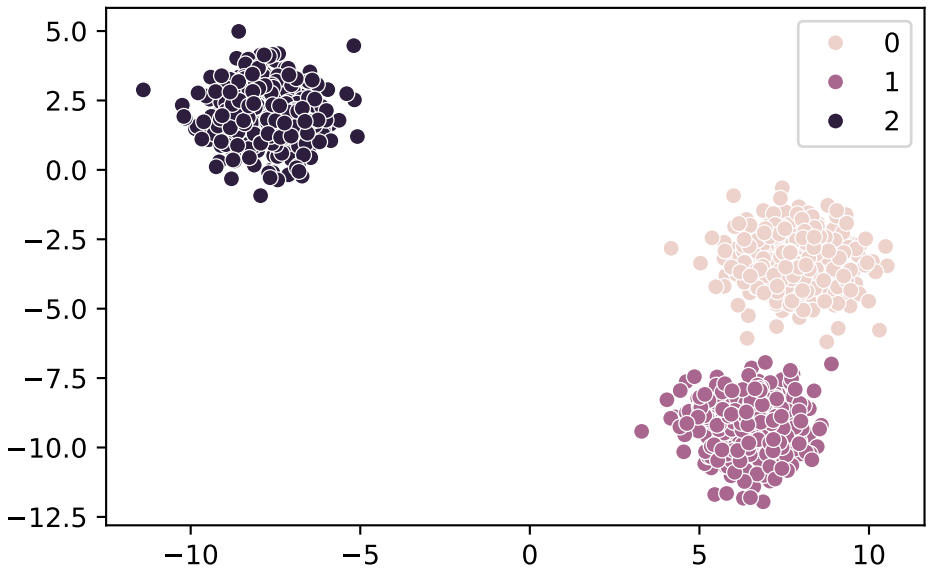
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.inspection import DecisionBoundaryDisplay

import pandas as pd
import seaborn as sns
from sklearn.datasets import make_blobs
```

5 Cargas datos

```
X, y = make_blobs(n_samples=1000, centers=3, random_state=6)
```

6 Visualizar los datos

```
sns.scatterplot(x=X[:,0],y=X[:,1], hue=y)
plt.show()
```



7 Se normaliza y se divide los datos

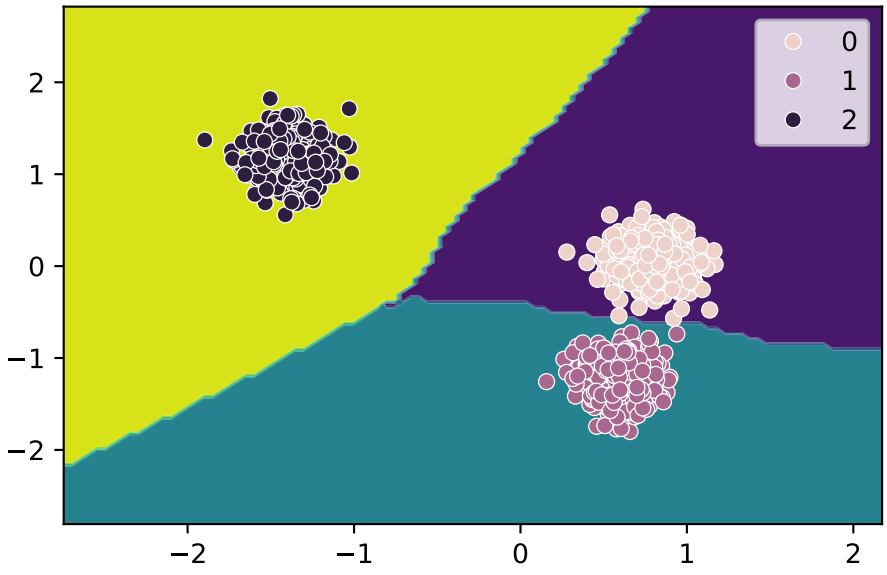
```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y)

# Scale the features using StandardScaler
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

8 Ajuste y evaluación del modelo

```
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# predecir con el modelo
y_pred = knn.predict(X_test)

# evaluarlo
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```
Accuracy: 1.0
```

```
DecisionBoundaryDisplay.from_estimator(knn, X_train)
sns.scatterplot(x=X[:,0],y=X[:,1], hue=y)
plt.show()
```



9 Anatomía de un algoritmo de aprendizaje

9.1 Bloques de construcción de un algoritmo de aprendizaje

En los algoritmos de aprendizaje que abarcamos en la sección anterior podemos observar los 3 bloques que se utilizan para construirlos:

Tabla 9.1: Bloques de un algoritmo de aprendizaje.

Bloque	Descripción
Función de pérdida	• Método para evaluar que tan bien se ajusta el modelo a los datos de entrenamiento • Si el modelo no predice adecuadamente, la función de pérdida da un resultado mayor
Criterio de optimización	• Debe estar basado en la función de pérdida
Rutina de optimización	• Utiliza los valores de la función objetivo y los datos para ajustar los parámetros del modelo

9.2 Descenso de gradiente (GD)

El *descenso de gradiente* es un algoritmo iterativo que permite encontrar el máximo o mínimo de una función dada, y se utiliza en los algoritmos de *machine learning (ML)* y *deep learning (DL)* para minimizar las funiciones de pérdida.

En conjunto con el *descenso de gradiente estocástico*, son de los algoritmos más utilizados en *ML* y *DL*.

9.2.1 Requisitos de la función a optimizar

- **Diferenciable:** tiene una derivada para cada punto en su dominio.

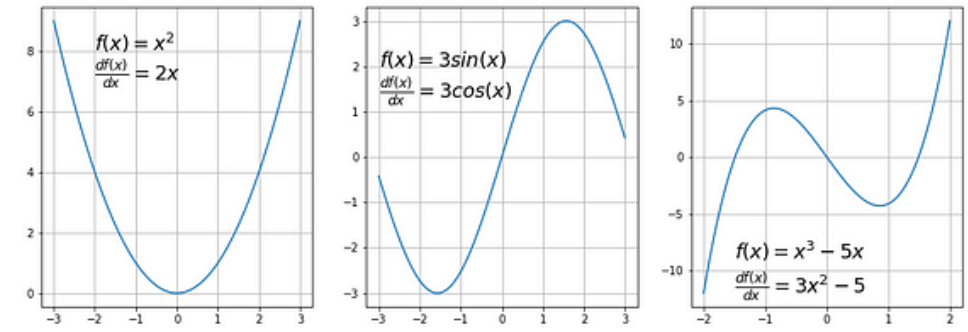


Figura 9.1: Funciones diferenciables

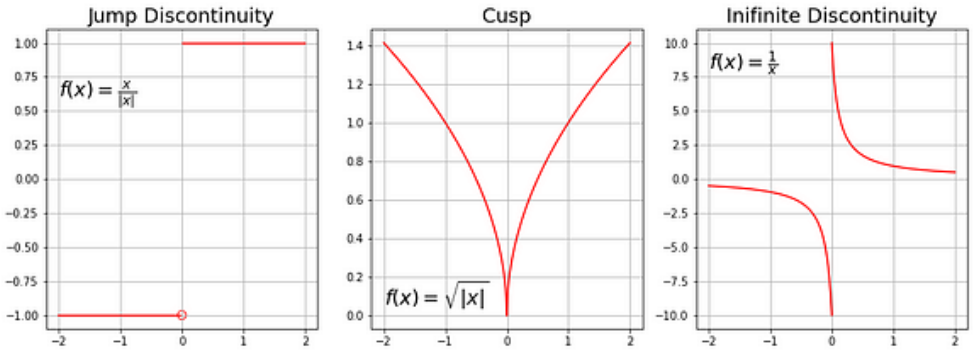


Figura 9.2: Funciones no diferenciables

- **Convexa:** para una función univariada, una línea que conecta dos puntos de la función pasa sobre o encima de la función. Las funciones convexas solo tienen un mínimo, que es el mínimo global.

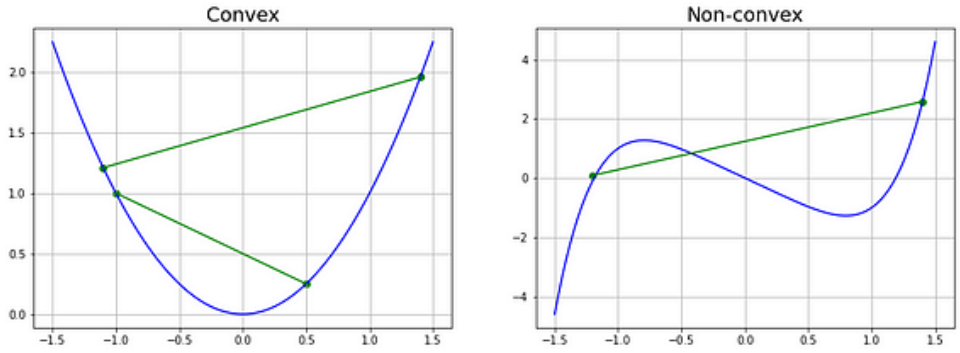


Figura 9.3: Funciones convexas

Los criterios de optimización de muchos modelos (regresión lineal y logística, SVM, entre otros) son convexos, por lo que el *GD* es un método adecuado.

Los criterios de optimización para las redes neuronales no son convexos (tienen mínimos locales y globales), pero en la práctica es suficiente encontrar mínimos globales, por lo que el *GD* también resulta un método útil.

9.2.2 Gradiente

Es la **pendiente** de una curva en una dirección específica.

En funciones univariadas la obtenemos evaluando la primera derivada en un punto de interés.

En funciones multivariadas, es un vector de derivadas en cada dirección principal, lo que conocemos como **derivadas parciales**.

El gradiente para una función $f(x)$ en un punto p está dado por:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

9.2.3 Algoritmo de descenso de gradiente

El método se puede resumir mediante la siguiente ecuación:

$$p_{n+1} = p_n - \alpha \nabla f(p_n)$$

Paso a paso:

1. Escoger un punto inicial: p_n
2. Calcular el gradiente en este punto: $\nabla f(p_n)$
3. Moverse en la dirección contraria al gradiente, a una distancia dada por la tasa de aprendizaje α
4. Repetir pasos 2 y 3 hasta que se cumpla lo siguiente:

- Número máximo de iteraciones alcanzado
- El tamaño del paso es más pequeño que la tolerancia definida (cambio en α o gradiente muy baja)

9.2.4 Imports

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

from typing import Callable
```

9.2.5 Ejemplo 1: función univariada, derivable no convexa

Función a optimizar:

$$f(x) = x^4 - 2x^3 + 2$$

Y su gradiente:

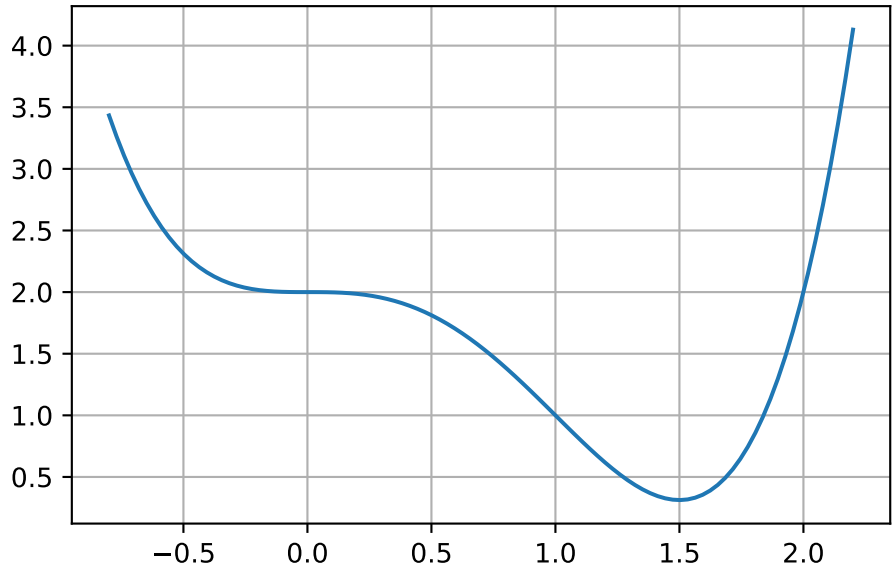
$$\frac{df(x)}{dx} = 4x^3 - 6x^2$$

Definiendo $f(x)$ y $\frac{df(x)}{dx}$ en python

```
# f(x)
def f_ej1(x:float):
  return x**4-2*x**3+2

# df(x)/dx
def dfdx_ej1(x:float):
  return 4*x**3-6*x**2
```

9.2.6 Gráfica de $f(x)$

```
x = np.linspace(-0.8, 2.2, 100)
y = f_ej1(x)
plt.grid(True)

plt.plot(x,y)
```



Definiendo algoritmo `gradient_descent`

```
def gradient_descent(start: float, gradient: Callable[[float], float],
                     learn_rate: float, max_iter: int, tol: float = 0.01):
    x = start
    steps = [start]  # history tracking

    for _ in range(max_iter):
        diff = learn_rate*gradient(x)
        if np.abs(diff) < tol:
            break
        x = x - diff
        steps.append(x)  # history tracing

    return steps, x
```

Llamando a `gradient_descent` para el ejemplo 1

```
start = 2
gradient = dfdx_ej1
learn_rate = 0.1
max_iter = 100

gradient_descent(start, gradient, learn_rate, max_iter)
```

```
([2, 1.2, 1.3728000000000002, 1.4686874222592001, 1.4957044497076786],
 1.4957044497076786)
```

Variaciones al ejemplo 1

9.2.7 Códigos usados para esta sección

```
# Gradient Descent
# gcorazzari

import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np

from typing import Callable


def update(frame):
    x = x_gd[:frame]
    y = y_gd[:frame]
    line_gd.set_xdata(x)
    line_gd.set_ydata(y)
    return line_gd


def gradient_descent(
    start: float,
    gradient: Callable[[float], float],
    learn_rate: float,
    max_iter: int,
    tol: float = 0.01,
```

```
):
    x = start
    steps = [start]  # history tracking

    for _ in range(max_iter):
        diff = learn_rate * gradient(x)
        if np.abs(diff) < tol:
            break
        x = x - diff
        steps.append(x)  # history tracing

    return steps, x


# EJ1-4
def f_ej1(x: float):
    return x**4 - 2 * x**3 + 2


def dfdx_ej1(x: float):
    return 4 * x**3 - 6 * x**2


start = -0.5
gradient = dfdx_ej1
learn_rate = 0.3
max_iter = 100

res_ej1 = gradient_descent(start, gradient, learn_rate, max_iter)

fig, ax = plt.subplots()

x = np.linspace(-0.8, 2.2, 100)
y = f_ej1(x)

line = ax.plot(x, y)

x_gd = np.array(res_ej1[0])
y_gd = f_ej1(x_gd)

line_gd = ax.plot(
    res_ej1[0][0], f_ej1(res_ej1[0][0]), "ro-", linewidth=0.5, markersize=2
```

```
)[0]

ani = animation.FuncAnimation(fig=fig, func=update, frames=102, interval=200)

plt.grid(True)
plt.title("Inicio=-0.5, alpha=0.3")

writer = animation.PillowWriter(fps=5)

plt.show()
ani.save("ej1-4.gif", writer=writer)


# EJ1-3
def f_ej1(x: float):
    return x**4 - 2 * x**3 + 2


def dfdx_ej1(x: float):
    return 4 * x**3 - 6 * x**2


start = -0.5
gradient = dfdx_ej1
learn_rate = 0.1
max_iter = 100

res_ej1 = gradient_descent(start, gradient, learn_rate, max_iter)

fig, ax = plt.subplots()

x = np.linspace(-0.8, 2.2, 100)
y = f_ej1(x)

line = ax.plot(x, y)

x_gd = np.array(res_ej1[0])
y_gd = f_ej1(x_gd)

line_gd = ax.plot(
    res_ej1[0][0], f_ej1(res_ej1[0][0]), "ro-", linewidth=0.5, markersize=2
)[0]
```

```
ani = animation.FuncAnimation(fig=fig, func=update, frames=9, interval=500)

plt.grid(True)
plt.title("Inicio=-0.5, alpha=0.1")

writer = animation.PillowWriter(fps=5)

plt.show()
ani.save("ej1-3.gif", writer=writer)


# EJ1-2
def f_ej1(x: float):
    return x**4 - 2 * x**3 + 2


def dfdx_ej1(x: float):
    return 4 * x**3 - 6 * x**2


start = 2
gradient = dfdx_ej1
learn_rate = 0.3
max_iter = 100

res_ej1 = gradient_descent(start, gradient, learn_rate, max_iter)

fig, ax = plt.subplots()

x = np.linspace(-0.8, 2.2, 100)
y = f_ej1(x)

line = ax.plot(x, y)

x_gd = np.array(res_ej1[0])
y_gd = f_ej1(x_gd)

line_gd = ax.plot(
    res_ej1[0][0], f_ej1(res_ej1[0][0]), "ro-", linewidth=0.5, markersize=2
)[0]

ani = animation.FuncAnimation(fig=fig, func=update, frames=4, interval=200)
```

```
plt.grid(True)
plt.title("Inicio=2, alpha=0.3")

writer = animation.PillowWriter(fps=5)

plt.show()
ani.save("ej1-2.gif", writer=writer)


# EJ1-1
def f_ej1(x: float):
    return x**4 - 2 * x**3 + 2


def dfdx_ej1(x: float):
    return 4 * x**3 - 6 * x**2


start = 2
gradient = dfdx_ej1
learn_rate = 0.1
max_iter = 100

res_ej1 = gradient_descent(start, gradient, learn_rate, max_iter)

fig, ax = plt.subplots()

x = np.linspace(-0.8, 2.2, 100)
y = f_ej1(x)

line = ax.plot(x, y)

x_gd = np.array(res_ej1[0])
y_gd = f_ej1(x_gd)

line_gd = ax.plot(
    res_ej1[0][0], f_ej1(res_ej1[0][0]), "ro-", linewidth=0.5, markersize=2
)[0]

ani = animation.FuncAnimation(fig=fig, func=update, frames=6, interval=500)

plt.grid(True)
```

```
plt.title("Inicio=2, alpha=0.1")

writer = animation.PillowWriter(fps=5)

plt.show()
ani.save("ej1-1.gif", writer=writer)
```

10 Capítulo 5

10.1 5.2:Selección de algoritmos

Elegir un algoritmo puede ser una tarea difícil. En el caso de que se disponga de mucho tiempo se pueden probar varios, sin embargo no siempre es el caso, por lo que hay una serie de preguntas que se pueden realizar con el fin de hacer el proceso más eficiente.

10.1.1 Explicabilidad o interpretabilidad

Con que facilidad el algoritmo logra explicar las predicciones que realiza, por lo general un modelo que realiza una predicción específica es dificíl de entender y aún más de explicar. Algunos ejemplos son los modelos de redes neuronales o el método de emsamble. Estos algoritmos que carecen de tal explicación se llaman algoritmo de caja negra.Por otro lado, los algoritmos de aprendizaje kNN, regresión lineal o árbol de decisión producen modelos que no siempre son los más precisos, sin embargo, la forma en que hacen su predicción es muy sencilla.

Ejemplo

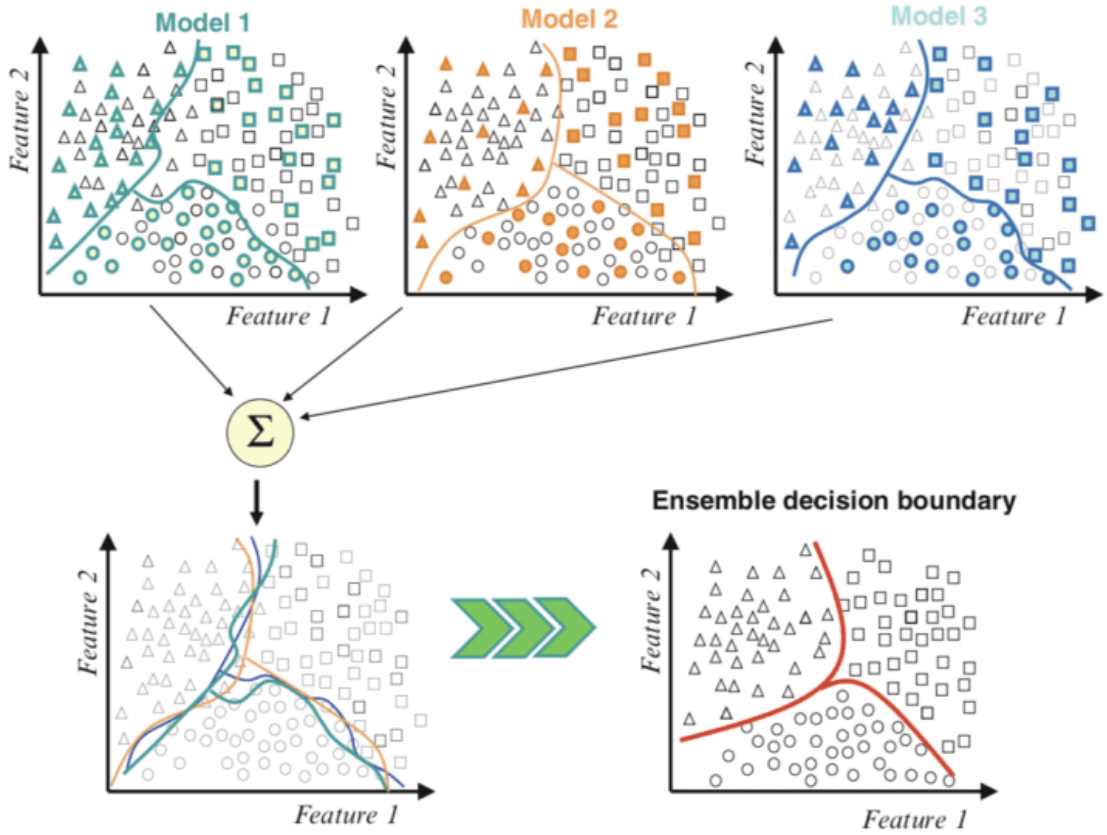


Figura 10.1: **Figura 1**. Método de emsamble

10.1.2 Requisitos de memoria

Si el conjunto de datos se puede cargar de forma completa en la memoria RAM del servidor o computador, entonces la disponibilidad de logaritmos es amplia. Sin embargo, si ese no es el caso, se debe optar por logaritmos de aprendizaje incremental, estos pueden mejorar el modelo añadiendo más datos gradualmente, básicamente se adaptan a nuevos nuevos sin el olvidar la información ya existente.

10.1.3 Número de funciones y característica

Algunos algoritmos, como las redes neuronales y el descenso de gradiente, pueden manejar un gran número de ejemplos y millones de características. Otros, como SVM, pueden ser muy modestos en su capacidad. Entonces, a la hora de escoger un logaritmo se debe considerar el tamaño de los datos y la cantidad de funciones.

10.1.4 Características categóricas frente a numéricas

Algunos algoritmos solo pueden funcionar con datos numéricos, por lo que si se tienen datos en un formato categórico o no numérico, se deberá considerar un proceso para convertirlos en datos numéricos mediante técnicas como la codificación one-hot.

10.1.5 linealidad de los datos

Si los datos son linealmente separables o pueden modelarse mediante un modelo lineal, se puede utilizar SVM, regresión logística o la regresión lineal, si no es el caso las redes neuronales o los algoritmos de conjunto, son una mejor opción.

Ejemplo

10.1.6 Velocidad de entrenamiento

Es el tiempo que tarda un algoritmo en aprender y crear un modelo. Las redes neuronales son conocidas por la considerable cantidad de tiempo que requieren para entrenar un modelo. Los algoritmos de máquina tradicionales como K-Vecinos más cercanos y Regresión logística toman mucho menos tiempo. Algunos algoritmos, como Bosque aleatorio, requieren diferentes tiempos de entrenamiento según los núcleos de CPU que se utilizan.

Ejemplo

10.1.7 Velocidad de predicción

Tiempo que le toma a un modelo hacer sus predicciones, en este caso se debe considerar qué tan rápido debe ser el modelo a la hora de generar predicciones y para que función se está utilizando el modelo escogido. Si no se quiere adivinar cuál es el mejor algoritmo para los datos, una forma de elegir es utilizar la prueba de validación.

Ejemplo

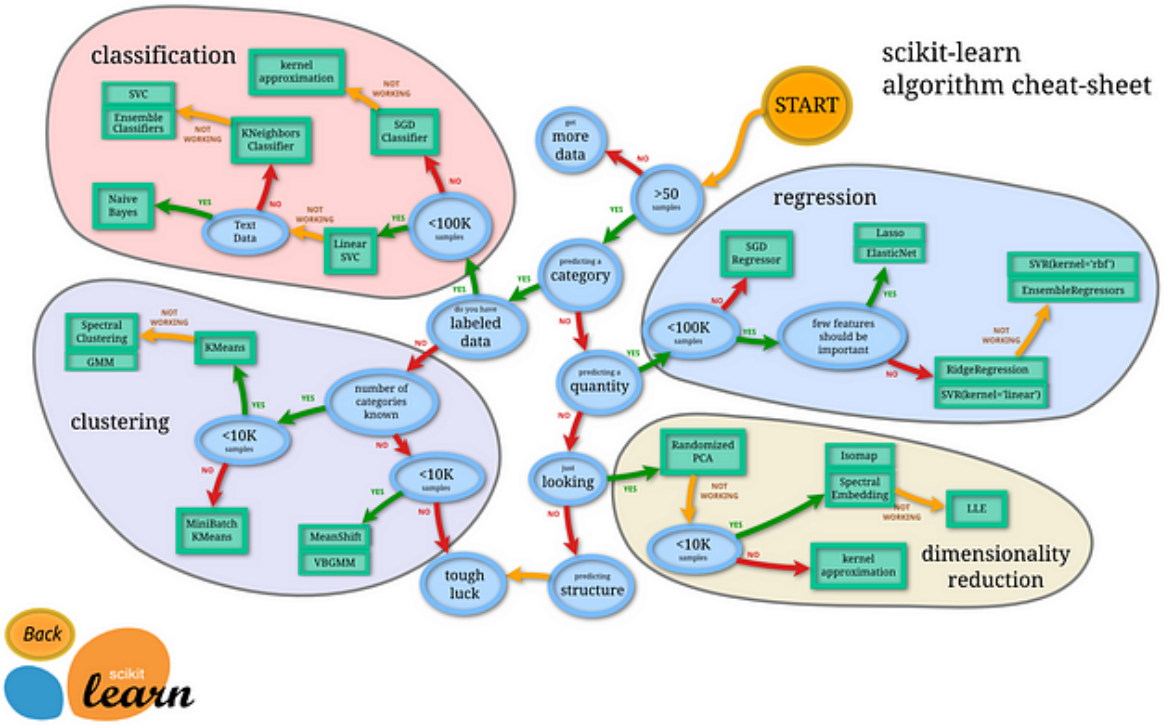


Figura 10.2: **Figura 6**. Diagrama Selección del algoritmo

10.2 5.3:Three sets

En Machine Learning, para el estudio y construcción de algoritmos que sean capaces de aprender de los datos y hacer predicciones utilizando esa información, se utiliza la construcción de un modelo matemático a partir de datos de entrada y estos datos para ser utilizados se dividen en conjuntos de datos, estos son:

1. Conjunto de entrenamiento
2. Conjunto de validación
3. Conjunto de prueba

El conjunto de entrenamiento, suele ser el conjunto más grande y es el que se utiliza para construir el modelo, los conjuntos de validación y prueba suelen tener el mismo tamaño y esto son menores que en el conjunto de entrenamiento , tanto los conjuntos de validación y prueba no son usados para construir el modelo. A estos conjuntos se les suele llamar conjuntos esfera.

Ejemplo

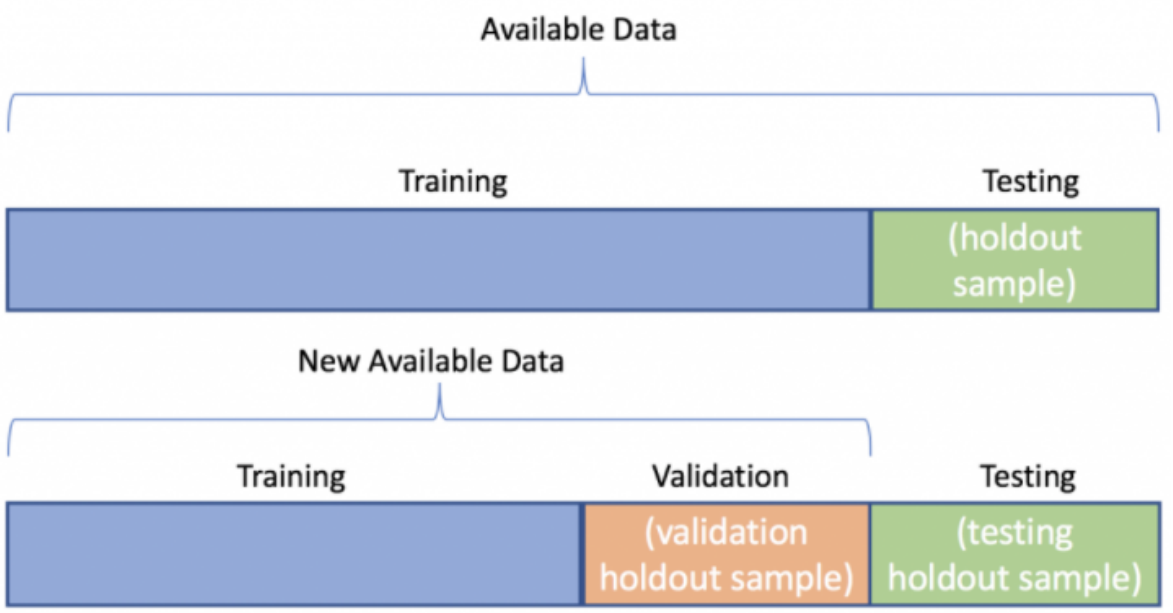


Figura 10.3: **Figura 8**. Solución problemas

11 Ajuste de hiperparámetros

El analista de datos debe seleccionar buenos hiperparámetros para el algoritmo que se esté trabajando. Como se sabe que lkos hiperparámetros no son optimizados por el algoritmo de aprendizaje por si solo; el analista debe ajustar la mejor combinación de hiperparámetros.

Una forma usual de realizar esto es, si se tienen suficientes datos para tener un conjunto de validación decente (cada calse tiene al menos un par de docenas de ejemplos) y el número de hiperparámetros y sus rangos no son muy grandes; se puede utilizar **grid search**. Este método es la más simple estrategia para ajustar los hiperparámetros.

Asuma que entrenamos un SVM y queremos ajustar dos hiperparámetros: el parámetro de penalización C (un número real positivo) y el kernel (si es 'linear' o 'rbf'). Si es la primera vez trabajando con el datset y no tenemos un rango de valores para C, un truco muy común es usar una escala logarítimica; entonces podemos tomar los valores de $[0.001, 0.01, 0.1, 1.0, 10, 100, 1000]$. Note que debido lo anterior tenemos 14 combinaciones de hiperparámetros diferentes: [(0.001, 'linear'), (0.01, 'linear'), (0.1, 'linear'), (1.0, 'linear'), (10, 'linear'), (100, 'linear'), (1000, 'linear'), (0.001, 'rbf'), (0.01, 'rbf'), (0.1, 'rbf'), (1.0, 'rbf'), (10, 'rbf'), (100, 'rbf'), (1000, 'rbf')]

Lo que se hace es tomar el conujunto de entrenamiento y entrenarlo con los 14 modelos diferentes. Luego, se evalúa el rendimiento de cada modelo con los datos de validación usando alguna métrica. Y se elige el que tenga la mejor métrica. Y por último se evalúa con el conjunto de prueba.

Un problema es el consumo de tiempo. Para ello, hay técnicas más eficientes como **random search** y **bayesian hyperparameter optimization**.

En el random search, uno le da una una distribución estadística para cada hiperparámetro y el número total de combinaciones que ser quieren realizar. La técnica bayesiana utiliza los resultados anteriores para elegir los próximos valores para evaluar.

11.1 Validación Cruzada

Cuando no se tienen muchos datos para ajustar los hiperparámetros. Entonces, podemos dividir el conjunto de entrenamiento en varios subconjuntos (*fold*) del mismo tamaño, lo usual es usar 5 folds. Así, se dividen los datos de entrenamiento en 5 folds $\{F_1, F_2, F_3, F_4, F_5\}$ cada una contiene el 20% de los datos de entrenamiento. Así, se entrenan 5 modelos, de la siguiente forma: para el primer modelo f_1 se utilizan los folds F_2, F_3, F_4, F_5 y F_1 se utiliza como conjunto

de validación; para el segundo modelo f_2 se utilizan los folds F_1, F_3, F_4, F_5 y el F_2 es el conjutno de validación; y aspi hasta completar f_5.

Se puede aplicar grid search con la validación cruzada para encontrar los mejores valores para los hiperparámetro de nuestro modelo.

11.2 Ejemplo

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
digits = datasets.load_digits()
n_samples = len(digits.images)
X = digits.images.reshape((n_samples, -1))
y = digits.target == 8

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=0)
```

```
import pandas as pd


def print_dataframe(filtered_cv_results):
    """Pretty print for filtered dataframe"""
    for mean_precision, std_precision, mean_recall, std_recall, params in zip(
        filtered_cv_results["mean_test_precision"],
        filtered_cv_results["std_test_precision"],
        filtered_cv_results["mean_test_recall"],
        filtered_cv_results["std_test_recall"],
        filtered_cv_results["params"],
    ):
        print(
            f"precision: {mean_precision:0.3f} (±{std_precision:0.03f}),"
            f" recall: {mean_recall:0.3f} (±{std_recall:0.03f}),"
            f" for {params}"
        )
    print()


def refit_strategy(cv_results):
    """Define the strategy to select the best estimator.
```

```
    The strategy defined here is to filter-out all results below a precision threshold
    of 0.98, rank the remaining by recall and keep all models with one standard
    deviation of the best by recall. Once these models are selected, we can select the
    fastest model to predict.

    Parameters
    ----------
    cv_results : dict of numpy (masked) ndarrays
        CV results as returned by the `GridSearchCV`.

    Returns
    -------
    best_index : int
        The index of the best estimator as it appears in `cv_results`.
    """
    # print the info about the grid-search for the different scores
    precision_threshold = 0.98

    cv_results_ = pd.DataFrame(cv_results)
    print("All grid-search results:")
    print_dataframe(cv_results_)

    # Filter-out all results below the threshold
    high_precision_cv_results = cv_results_[
        cv_results_["mean_test_precision"] > precision_threshold
    ]

    print(f"Models with a precision higher than {precision_threshold}:")
    print_dataframe(high_precision_cv_results)

    high_precision_cv_results = high_precision_cv_results[
        [
            "mean_score_time",
            "mean_test_recall",
            "std_test_recall",
            "mean_test_precision",
            "std_test_precision",
            "rank_test_recall",
            "rank_test_precision",
            "params",
        ]
    ]
```

```
    # Select the most performant models in terms of recall
    # (within 1 sigma from the best)
    best_recall_std = high_precision_cv_results["mean_test_recall"].std()
    best_recall = high_precision_cv_results["mean_test_recall"].max()
    best_recall_threshold = best_recall - best_recall_std

    high_recall_cv_results = high_precision_cv_results[
        high_precision_cv_results["mean_test_recall"] > best_recall_threshold
    ]
    print(
        "Out of the previously selected high precision models, we keep all the\n"
        "the models within one standard deviation of the highest recall model:"
    )
    print_dataframe(high_recall_cv_results)

    # From the best candidates, select the fastest model to predict
    fastest_top_recall_high_precision_index = high_recall_cv_results[
        "mean_score_time"
    ].idxmin()

    print(
        "\nThe selected final model is the fastest to predict out of the previously\n"
        "selected subset of best models based on precision and recall.\n"
        "Its scoring time is:\n\n"
        f"{high_recall_cv_results.loc[fastest_top_recall_high_precision_index]}"
    )

    return fastest_top_recall_high_precision_index
```

Ahora, se define el grid search

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
scores = ['precision', 'recall']

tuned_params = [
    {'kernel':['rbf'], 'gamma':[1e-3, 1e-4], 'C':[1, 10, 100, 1000]},
    {'kernel':['linear'], 'C':[1, 10, 100, 1000]}
]

grid_search = GridSearchCV(SVC(), tuned_params, scoring=scores, refit=refit_strategy)
grid_search.fit(X_train, y_train)
```

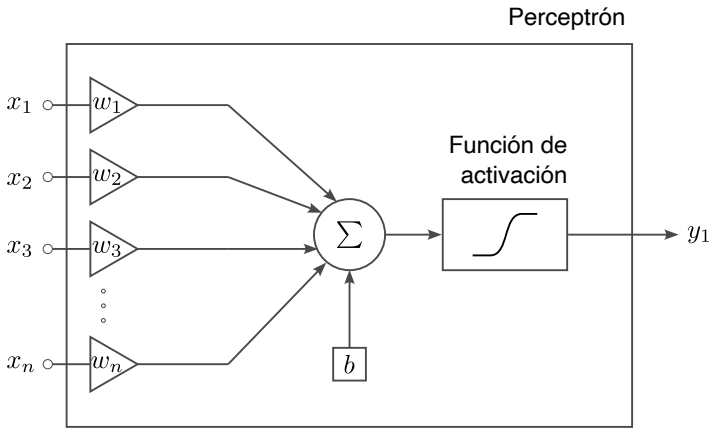
```
All grid-search results:
precision: 1.000 (±0.000), recall: 0.854 (±0.063), for {'C': 1, 'gamma': 0.001, 'kernel': 'rb
precision: 1.000 (±0.000), recall: 0.257 (±0.061), for {'C': 1, 'gamma': 0.0001, 'kernel': '
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 10, 'gamma': 0.001, 'kernel': '
precision: 0.968 (±0.039), recall: 0.780 (±0.083), for {'C': 10, 'gamma': 0.0001, 'kernel':
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 100, 'gamma': 0.001, 'kernel':
precision: 0.905 (±0.058), recall: 0.889 (±0.074), for {'C': 100, 'gamma': 0.0001, 'kernel':
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 1000, 'gamma': 0.001, 'kernel':
precision: 0.904 (±0.058), recall: 0.890 (±0.073), for {'C': 1000, 'gamma': 0.0001, 'kernel'
precision: 0.695 (±0.073), recall: 0.743 (±0.065), for {'C': 1, 'kernel': 'linear'}
precision: 0.643 (±0.066), recall: 0.757 (±0.066), for {'C': 10, 'kernel': 'linear'}
precision: 0.611 (±0.028), recall: 0.744 (±0.044), for {'C': 100, 'kernel': 'linear'}
precision: 0.618 (±0.039), recall: 0.744 (±0.044), for {'C': 1000, 'kernel': 'linear'}

Models with a precision higher than 0.98:
precision: 1.000 (±0.000), recall: 0.854 (±0.063), for {'C': 1, 'gamma': 0.001, 'kernel': 'rb
precision: 1.000 (±0.000), recall: 0.257 (±0.061), for {'C': 1, 'gamma': 0.0001, 'kernel': '
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 10, 'gamma': 0.001, 'kernel': '
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 100, 'gamma': 0.001, 'kernel':
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 1000, 'gamma': 0.001, 'kernel':

Out of the previously selected high precision models, we keep all the
the models within one standard deviation of the highest recall model:
precision: 1.000 (±0.000), recall: 0.854 (±0.063), for {'C': 1, 'gamma': 0.001, 'kernel': 'rb
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 10, 'gamma': 0.001, 'kernel': '
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 100, 'gamma': 0.001, 'kernel':
precision: 1.000 (±0.000), recall: 0.877 (±0.069), for {'C': 1000, 'gamma': 0.001, 'kernel':


The selected final model is the fastest to predict out of the previously
selected subset of best models based on precision and recall.
Its scoring time is:

mean_score_time                                         0.008195
mean_test_recall                                        0.877206
std_test_recall                                         0.069196
mean_test_precision                                          1.0
std_test_precision                                           0.0
rank_test_recall                                               3
rank_test_precision                                            1
```

```
params                 {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'}
Name: 2, dtype: object


GridSearchCV(estimator=SVC(),
             param_grid=[{'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']},
                         {'C': [1, 10, 100, 1000], 'kernel': ['linear']}],
             refit=<function refit_strategy at 0x130c30900>,
             scoring=['precision', 'recall'])
```

12 Redes neurales

12.1 Perceptrón y redes neuronales

El perceptrón en el modelo más secillo de las neuronas. Se le llama también *neurona artificial*.



La idea es que cada uno de los *features* se multiplica por un peso w_k, se le suma un *bias* b y al resultado de esta operación se le aplica la función de activación, que finalmente produce la salida y. Esta función de activación preferiblemente debe ser diferenciable y tres de las funciones más comunes son la función logística, la función tangente hiperbólica y la función lineal rectificada unitaria (ReLU):

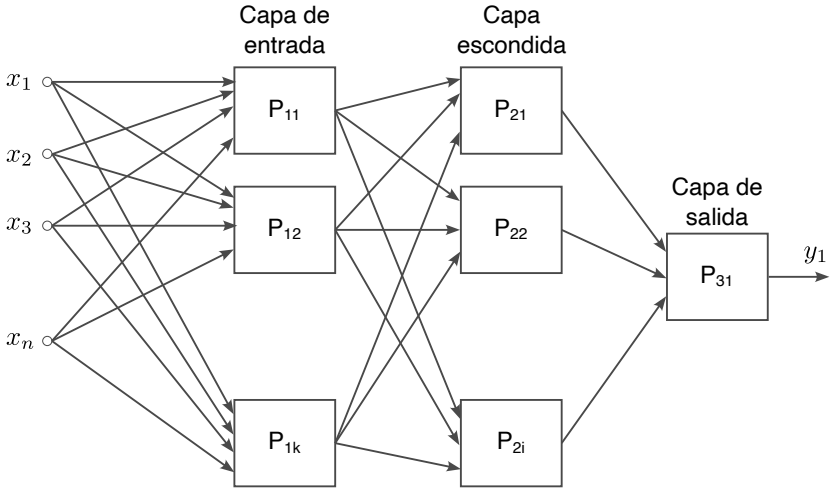
$$\text{logística: } f(x) = \frac{1}{1+e^{-x}}$$

$$\text{tangente hiperbólica: } f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$\text{ReLU: } f(x) = \begin{cases} 0, & \text{si } x < 0 \\ x, & \text{en otro caso} \end{cases}$$

En sí, el perceptrón es clasificador muy similar a la regresión logística. Sin embargo, es muy poco común que se utilice un solo perceptrón, sino que se utiliza varias capas (*layers*) de múltiples perceptrones, de manera que se pueda clasificar casos más complejos, a esto se le llama Perceptrón de capas múltiples (*Multiple Layer Perceptron*, MLP).

Una capa está compuesta por varios perceptrones que están conectados con las entradas o con las salidas de la capa anterior, tal y como se muestra en la figura siguiente:



cada uno de los bloques P_{ij} es un perceptrón. En la figura, solo se tiene una salida, pero bien podría tenerse más, por lo que la capa de salida podría tener más de un perceptrón. De igual manera, se puede tener más de una capa escondida.

Lo que se debe hacer ahora es que, a partir de los datos que se tienen, encontrar los valores de los pesos w_k y b_k, es decir, entrenar a la red.

12.2 Entrenamiento de la red neuronal

Para entrenar la red, se debe buscar los valores. Para ello se minimiza una función de costo, como por ejemplo el error cuadrático medio.

Para lograr la optimización se suele utilizar el descenso del gradiente, en un algoritmo llamado *Backpropagation*. Con el Backpropagation se calcula el gradiente de la función de costo con respecto a los pesos de la red, de una manera eficiente. Se calculan el gradiente una capa a la vez , iterando hacia atrás desde la última capa.

12.3 Comandos en python

Con la libraría scikit se puede crear y entrenar fácilmente una red neuronal.

```
from sklearn.neural_network import MLPClassifier

# Acá se cargan los datos
X = [[0., 0.], [1., 1.]] # acá se pondría una lista de listas con los featrures
Y = [0, 1] # acá los labels

# Acá se crea la red
clf = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(5, 2), random_state=1)

# Acá se entrena la red
clf.fit(X, Y)

# Y acá se utiliza la red para predecir el valor de la salida para una nueva entrada
clf.predict([[-1., -2.]])
```

```
array([0])
```

en la variable **hidden_layer_sizes** se pone el número de perceptrones para cada una de las capas escondidas. **alpha** es la fuerza del término de regularización L2. El solver es el algoritmo de optimización:

- `lbfgs` es un optimizador de la familia de métodos cuasi-Newton.
- `sgd` se refiere al descenso estocástico del gradiente.
- `adam` se refiere al optimizador estocástico del gradiente propuesto por Kingma, Diederik, and Jimmy Ba.

13 Problemas y soluciones

13.1 Regresión de Kernel

La regresión de Kernel es utilizada para los casos, que la entrada es un vector de características D-dimensional, con D > 3. La regresión del kernel es un método no paramétrico. Eso significa que no hay parámetros que aprender. El modelo se basa en los datos mismos (como en kNN). En su forma más simple, en la regresión del kernel buscamos un modelo como este:

$$f(x) = \frac{1}{N}\sum_{i=1}^{N} w_i y_i$$

, donde

$$w_i = \frac{Nk(\frac{x_i - x}{b}))}{\sum_{i=1}^{N} k(\frac{x_k - x}{b})}$$

La función k($\cdot$) es un núcleo(kernel). Puede tener diferentes formas, la más utilizada es el núcleo gaussiano:

$$k(z) = \frac{1}{\sqrt{2\pi}} exp(\frac{-z^2}{2}).$$

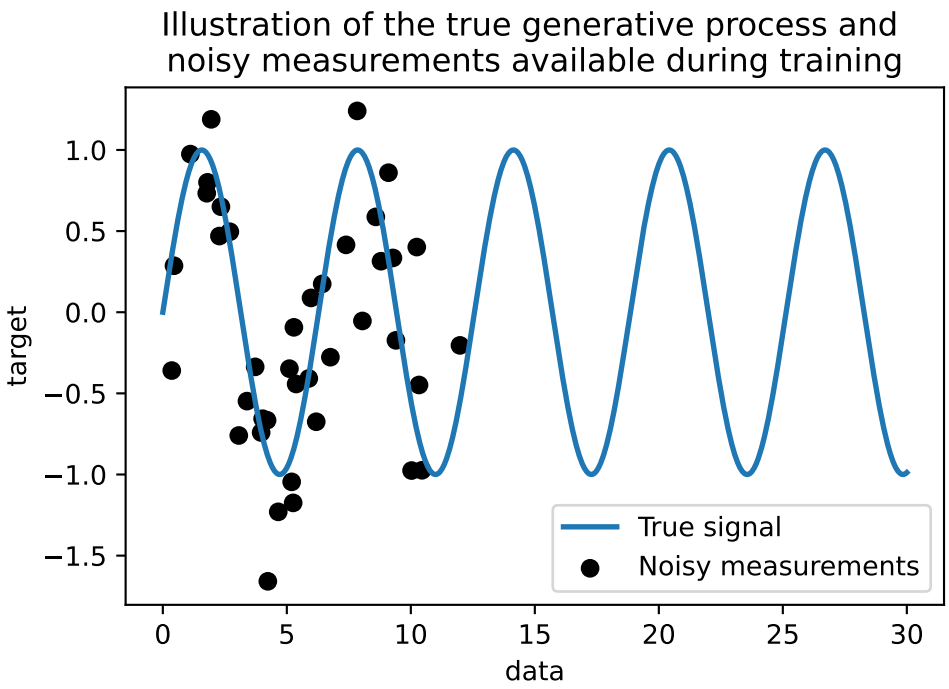
El valor b es un hiperparámetro que ajustamos usando el conjunto de validación.

Ejemplo 13.1.

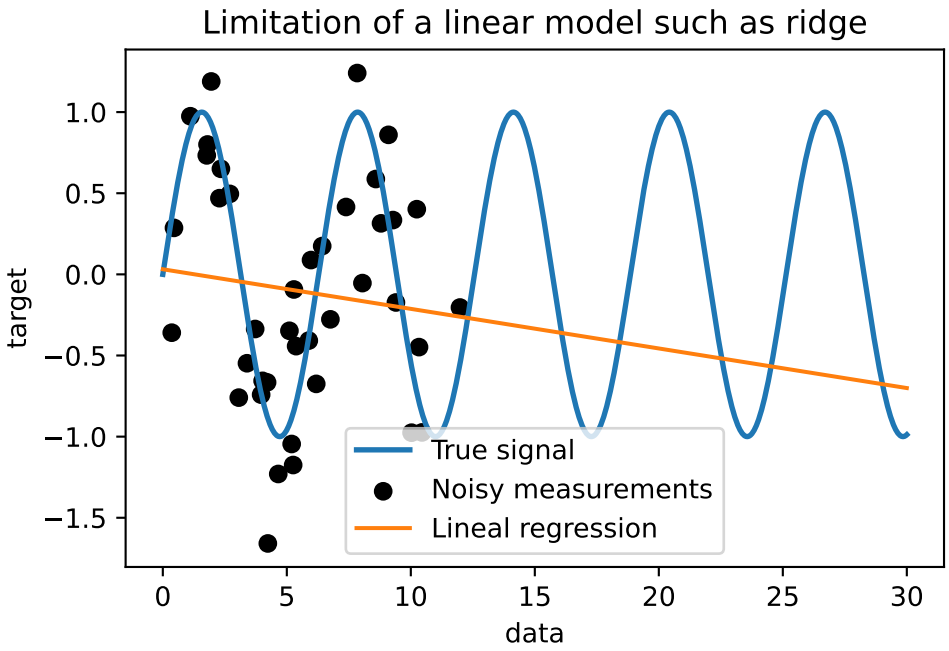
```
import numpy as np

rng = np.random.RandomState(0)
data = np.linspace(0, 30, num=1_000).reshape(-1, 1)
target = np.sin(data).ravel()
```

```
training_sample_indices = rng.choice(np.arange(0, 400), size=40, replace=False)
training_data = data[training_sample_indices]
training_noisy_target = target[training_sample_indices] + 0.5 * rng.randn(
    len(training_sample_indices)
)
```

```
import matplotlib.pyplot as plt

plt.plot(data, target, label="True signal", linewidth=2)
plt.scatter(
    training_data,
    training_noisy_target,
    color="black",
    label="Noisy measurements",
)
plt.legend()
plt.xlabel("data")
plt.ylabel("target")
_ = plt.title(
    "Illustration of the true generative process and \n"
    "noisy measurements available during training"
)
```



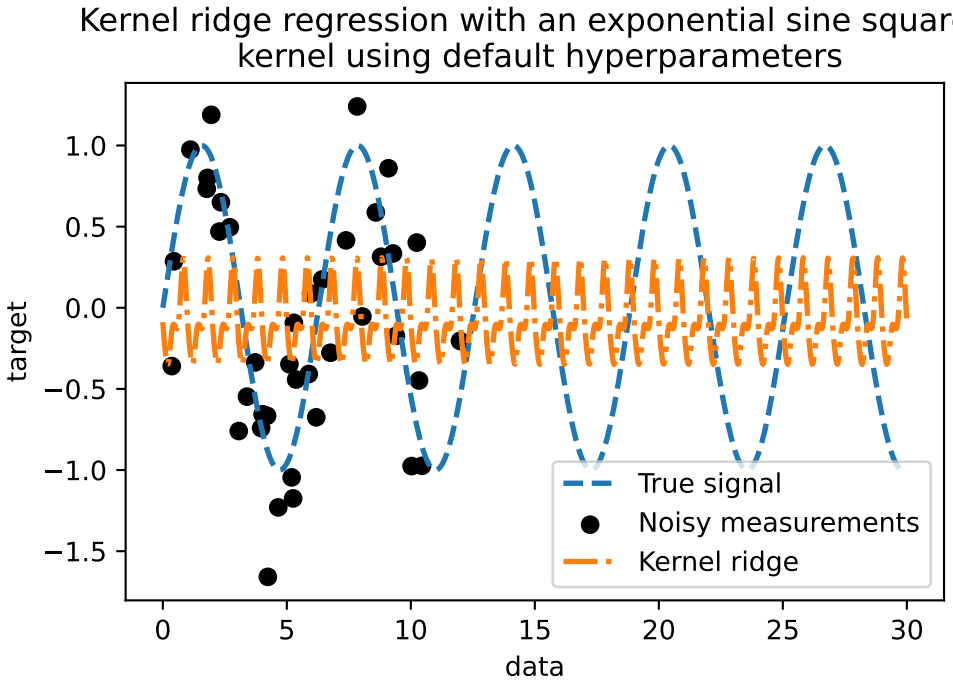

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(training_data, training_noisy_target)

plt.plot(data, target, label="True signal", linewidth=2)
plt.scatter(
    training_data,
    training_noisy_target,
    color="black",
    label="Noisy measurements",
)
plt.plot(data, ridge.predict(data), label="Lineal regression")
plt.legend()
plt.xlabel("data")
plt.ylabel("target")
_ = plt.title("Limitation of a linear model such as ridge")
```




```
import time

from sklearn.gaussian_process.kernels import ExpSineSquared
from sklearn.kernel_ridge import KernelRidge

kernel_ridge = KernelRidge(kernel=ExpSineSquared())

start_time = time.time()
kernel_ridge.fit(training_data, training_noisy_target)
print(
    f"Fitting KernelRidge with default kernel: {time.time() - start_time:.3f} seconds"
)

plt.plot(data, target, label="True signal", linewidth=2, linestyle="dashed")
plt.scatter(
    training_data,
    training_noisy_target,
    color="black",
    label="Noisy measurements",
)
plt.plot(
    data,
    kernel_ridge.predict(data),
```

```
    label="Kernel ridge",
    linewidth=2,
    linestyle="dashdot",
)
plt.legend(loc="lower right")
plt.xlabel("data")
plt.ylabel("target")
_ = plt.title(
    "Kernel ridge regression with an exponential sine squared\n "
    "kernel using default hyperparameters"
)
```

```
Fitting KernelRidge with default kernel: 0.002 seconds
```



Este modelo ajustado no es exacto. De hecho, no configuramos los parámetros del kernel y en su lugar utilizamos los predeterminados. Podemos inspeccionarlos.

```
kernel_ridge.kernel
```

```
ExpSineSquared(length_scale=1, periodicity=1)
```

```
from scipy.stats import loguniform

from sklearn.model_selection import RandomizedSearchCV

param_distributions = {
    "alpha": loguniform(1e0, 1e3),
    "kernel__length_scale": loguniform(1e-2, 1e2),
    "kernel__periodicity": loguniform(1e0, 1e1),
}
kernel_ridge_tuned = RandomizedSearchCV(
    kernel_ridge,
    param_distributions=param_distributions,
    n_iter=500,
    random_state=0,
)
start_time = time.time()
kernel_ridge_tuned.fit(training_data, training_noisy_target)
print(f"Time for KernelRidge fitting: {time.time() - start_time:.3f} seconds")
```

```
Time for KernelRidge fitting: 5.317 seconds
```

Ajustar el modelo ahora es más costoso desde el punto de vista computacional ya que tenemos que probar varias combinaciones de hiperparámetros. Podemos echar un vistazo a los hiperparámetros encontrados para hacer algunas intuiciones.

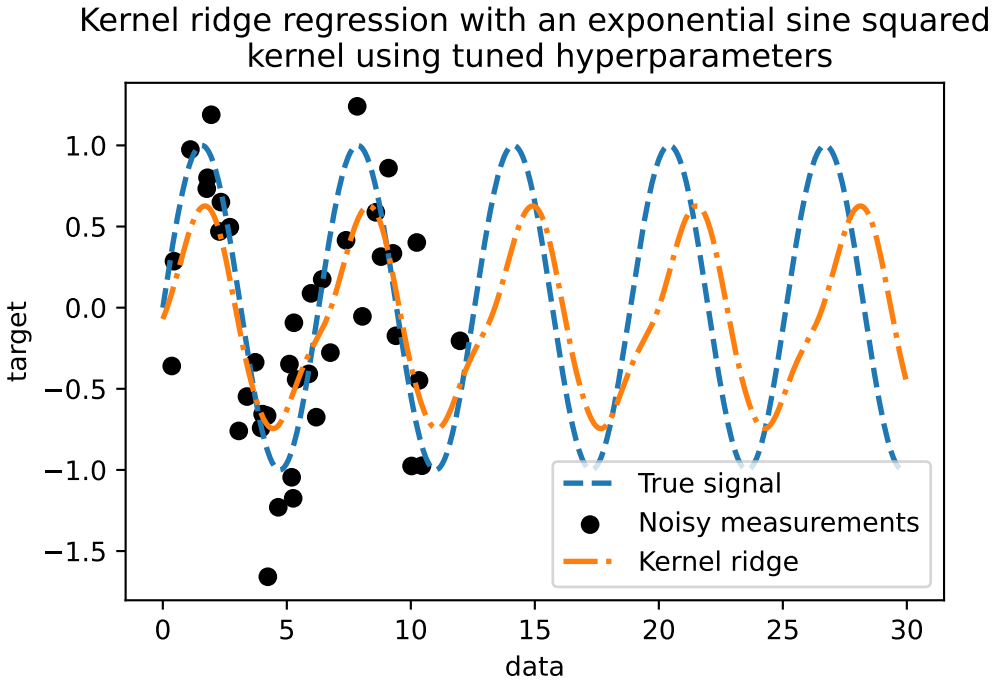
```
kernel_ridge_tuned.best_params_
```

```
{'alpha': 1.991584977345022,
 'kernel__length_scale': 0.7986499491396734,
 'kernel__periodicity': 6.6072758064261095}
```

```
start_time = time.time()
predictions_kr = kernel_ridge_tuned.predict(data)
print(f"Time for KernelRidge predict: {time.time() - start_time:.3f} seconds")

plt.plot(data, target, label="True signal", linewidth=2, linestyle="dashed")
plt.scatter(
    training_data,
    training_noisy_target,
    color="black",
```

```
    label="Noisy measurements",
)
plt.plot(
    data,
    predictions_kr,
    label="Kernel ridge",
    linewidth=2,
    linestyle="dashdot",
)
plt.legend(loc="lower right")
plt.xlabel("data")
plt.ylabel("target")
_ = plt.title(
    "Kernel ridge regression with an exponential sine squared\n "
    "kernel using tuned hyperparameters"
)
```

```
Time for KernelRidge predict: 0.003 seconds
```



Fuente: https://scikit-learn.org/stable/auto_examples/gaussian_process/plot_compare_gpr_krr.html#sphx-glr-auto-examples-gaussian-process-plot-compare-gpr-krr-py

13.2 Clasificación multiclase

La clasificación multiclase se refiere a aquellos casos en los que los datos contienen etiquetas que pertenecen a una de las C clases:

$$y \in \{1, ..., C\}$$

Por ejemplo, se puede clasificar utilizando features extraídos de un set de imágenes de frutas. En este ejemplo las etiquetas `y` serían:

```
y = ["manzana", "pera", "naranja"]
```

Cada imagen es una muestra y puede ser clasificada como **una** de las tres posibles clases. La clasificación multiclase asume que cada muestra está asociada a **una y solo una de las etiquetas**.

En el ejemplo, una fotografía no podría ser una pera y una naranja al mismo tiempo. Si esto no se cumple estaríamos ante un ejemplo de clasificación multietiqueta, que se verá más adelante.

Existen algunos algoritmos de clasificación que se pueden extender para ser algoritmos de clasificación multiclase:

- ID3 y otros algoritmos de árboles de decisión
- Regresión logística reemplazando la función sigmoidal con la función softmax
- kNN

Hay otros algoritmos que no se pueden extender a clasificación multiclase de forma simple, o en algunos casos, son mucho más eficientes en el caso de clasificación binaria. Ante esta situación, una estrategia común es llamada **uno versus el resto (OVR)**.

13.2.1 Uno versus el resto (OVR)

La idea detrás del enfoque de OVR es separar el problema de clasificación multiclase en múltiples casos de separación binaria.

En la siguiente figura podemos observar una ilustración con dos tipos de problemas de clasificación: binaria y multiclase.

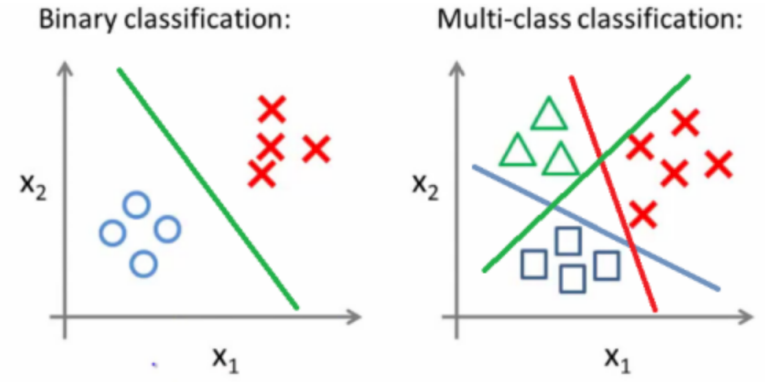


Figura 13.1: Ilustración de problemas de clasificación

Para la imagen de la derecha, un ejemplo de clasificación multiclase, podemos utilizar la estrategia de OVR, tal y como se muestra en la siguiente figura.

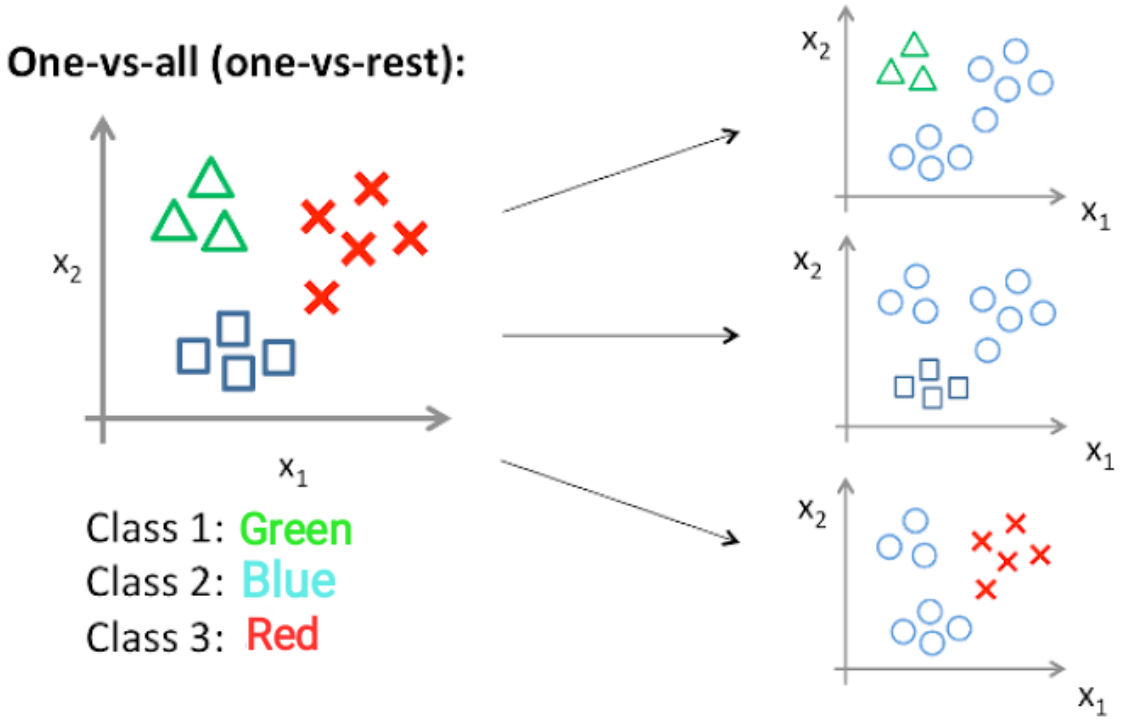


Figura 13.2: Ilustración de la clasificación multiclase

13.2.2 Implementación en python

Imports:

```
import pandas as pd
import seaborn as sns

import numpy as np
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

from sklearn.datasets import make_classification

import time
```

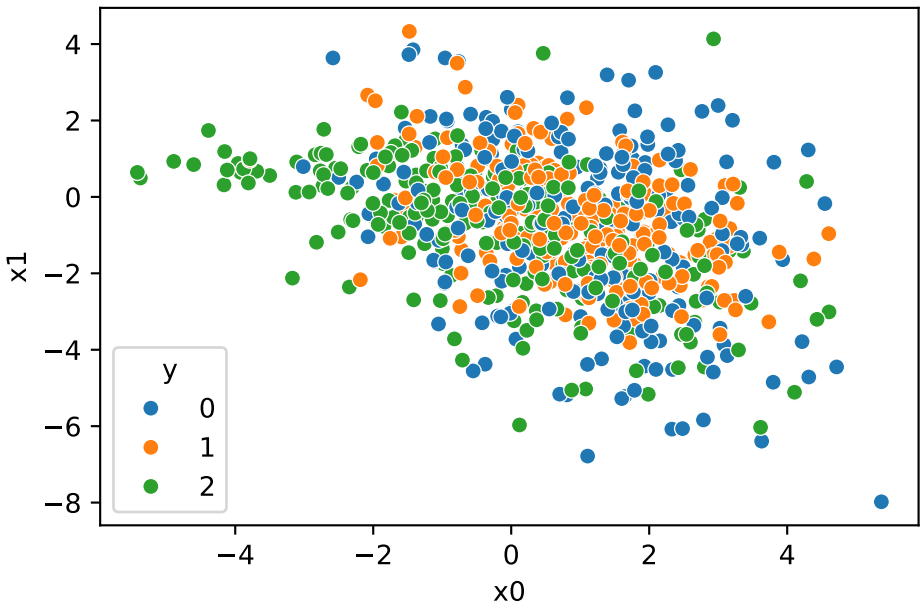
Creando datos aleatorios para clasificación:

```
# Generando un array aleatorio de 1000 muestras, 10 features y una etiqueta de y=[0,1,2]
X, y = make_classification(n_samples=1000, n_features=10, n_informative=5, n_redundant=5, n_

# Separando el array en conjuntos de prueba (25 %) y entrenamiento (75 %)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Colocando en dataframes para facilidad de presentación y graficación
df_train = pd.DataFrame({"y":y_train, "x0":X_train[:,0], "x1":X_train[:,1], "x2":X_train[:,2]

df_test = pd.DataFrame({"y":y_test, "x0":X_test[:,0], "x1":X_test[:,1], "x2":X_test[:,2], "x3

df_train
```

	y	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9				
0	2	-0.730468	0.397713	2.074811	3.268605	2.406136	-1.596750	-1.097254	0.839374	0.296
1	2	3.365775	-1.418898	1.872383	-0.693364	-3.133404	-2.877166	0.404063	-1.955554	-1.324
2	2	-3.921299	0.641428	-2.026409	-2.679728	2.478175	4.280245	0.959757	-1.031408	2.619
3	2	-0.476039	-0.361355	0.966933	2.514187	3.982157	-1.931150	-0.354184	2.025803	-0.295
4	0	-0.119220	-0.421085	0.706323	0.561597	-0.483120	0.058308	-0.852335	-0.952946	-0.161
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
745	2	-0.707565	-4.271104	1.374571	-0.328230	6.264860	-2.072022	0.239431	-1.271852	-0.637
746	1	-0.514813	-0.473396	-2.421689	-1.699244	-0.952055	2.352776	-0.218440	0.078120	-0.600
747	0	2.273762	1.123171	0.927239	2.014521	-1.140324	-2.785479	0.158036	2.155000	-0.954

			y	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9
748	0	5.365061	-7.979364	3.641793	0.967290	3.271363	-8.028106	-0.642597	-1.656500	-5.25
749	1	0.202570	1.402667	1.272342	3.307607	-1.209962	-0.738314	-1.770443	1.023892	-0.36

```
ax = sns.scatterplot(data=df_train, x="x0", y="x1", hue="y", palette="tab10")
```



Definiendo el modelo OVR

```
# Se define el modelo a usar dentro del OVR, en este caso SVC, puede ser logistico u otro
modelo = SVC()
# Entrenando el modelo con los datos de entrenamiento
clasificador = OneVsRestClassifier(modelo).fit(X_train, y_train)
```

Probando el modelo OVR `clasificador`

```
# Probando el modelo con los datos de prueba
prediccion = clasificador.predict(X_test)

# Colocando los datos predichos en un dataframe
df_pred = pd.DataFrame({"y":prediccion, "x0":X_test[:,0], "x1":X_test[:,1], "x2":X_test[:,2]
```

¿Cuál es el resultado del modelo `clasificador`?

Nos da un vector con las etiquetas predichas:

```
print(prediccion)
```

```
[2 0 2 2 0 0 0 1 0 2 2 2 2 0 1 2 1 0 0 0 1 2 2 2 2 1 0 1 0 0 1 0 1 2 0 1 1
 0 1 2 0 1 0 2 2 0 0 0 2 2 1 1 1 0 2 2 1 0 2 0 0 0 0 2 2 1 1 2 2 1 1 1 2 0
 0 0 1 0 0 2 1 2 0 1 1 0 1 0 2 2 0 2 1 2 2 0 2 2 1 0 1 2 1 2 2 0 2 1 1 0 2
 0 0 0 2 2 2 2 0 2 0 0 1 2 2 2 0 1 0 0 0 2 0 0 0 0 1 0 1 2 0 1 0 1 2 2 1 2
 1 0 2 2 1 1 2 1 1 0 1 2 2 0 2 0 0 1 0 0 1 0 1 0 1 0 2 1 2 0 0 0 0 1 0 1 0
 2 0 2 2 0 0 1 0 2 0 0 1 2 0 2 0 1 2 2 1 2 1 2 2 2 2 0 1 1 2 2 1 0 2 0 2 2
 0 0 1 0 0 0 1 0 1 1 1 0 2 1 0 0 0 0 1 2 1 1 1 0 0 1 1 2]
```

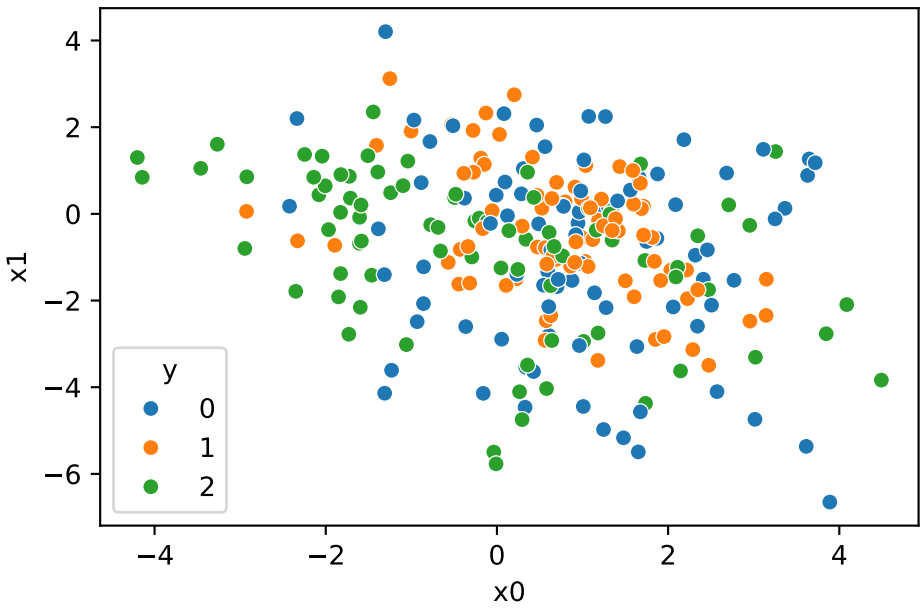
Y con el método `score(X, y)` podemos obtener la *mean accuracy*, que es una métrica bastante exigente pues requiere que para cada muestra la etiqueta sea asignada correctamente:

```
print("mean accuracy = ",clasificador.score(X_test, y_test))
```

```
mean accuracy =  0.908
```

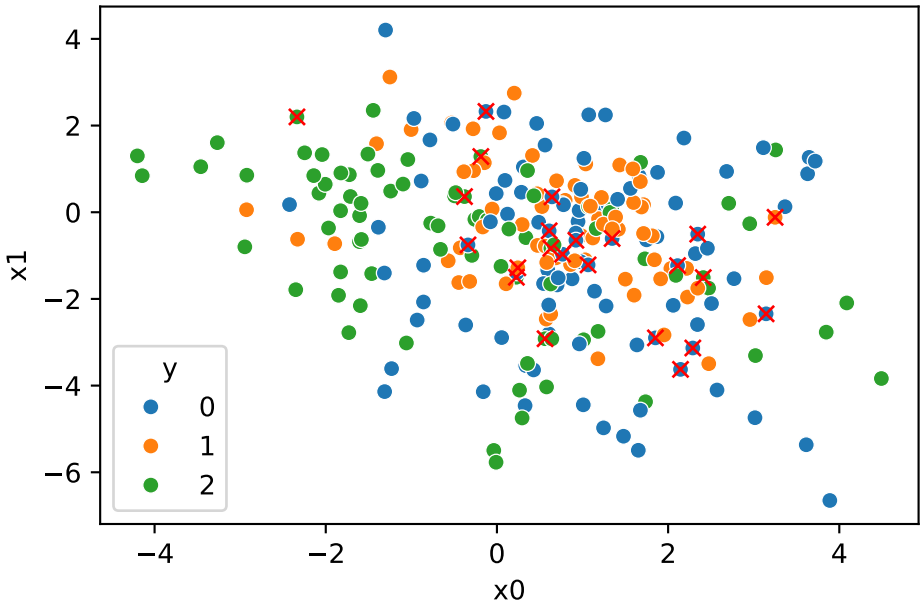
Luego, podemos comparar la gráfica del conjunto de prueba `df_test`, utilizando por ejemplo los features x_0 y x_1:

```
ax2 = sns.scatterplot(data=df_test, x="x0", y="x1", hue="y", palette="tab10")
```



Con la gráfica utilizando los mismos features x_0 y x_1 del conjunto de prueba pero esta vez con las etiquetas predichas por el modelo. Se marca una x roja sobre los puntos que fueron clasificados incorrectamente:

```
ax2 = sns.scatterplot(data=df_pred, x="x0", y="x1", hue="y", palette="tab10")
# Compara los y de prueba vs. los y predichos para marcar los que no se clasificaron correcta
for i in range(len(prediccion)):
    if prediccion[i] != y_test[i]:
        ax2.plot(df_pred["x0"].iloc[i], df_pred["x1"].iloc[i], "rx")
```

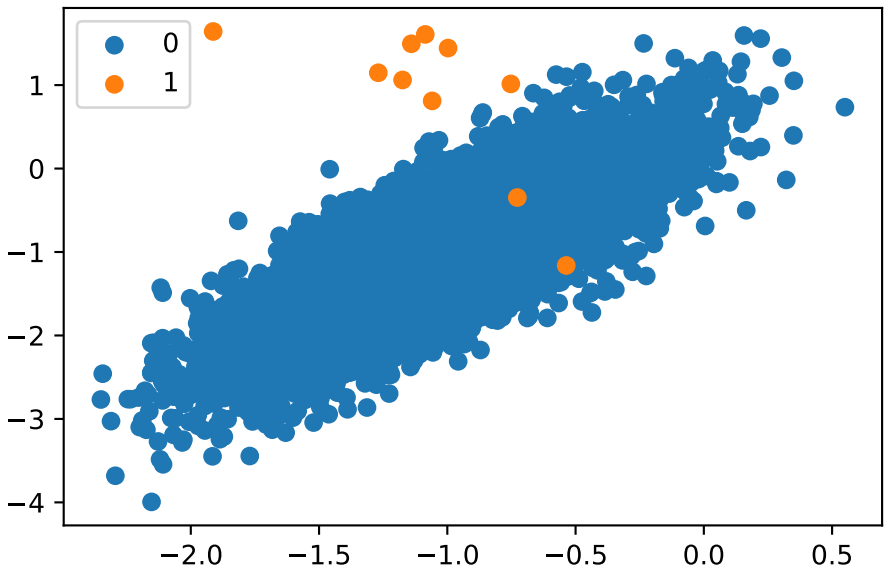


13.3 Clasificación con una clase

```
# Generate and plot a synthetic imbalanced classification dataset
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where

# define dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
 n_clusters_per_class=1, weights=[0.999], flip_y=0, random_state=4)
```

```
# summarize class distribution
counter = Counter(y)
print(counter)
# scatter plot of examples by class label
for label, _ in counter.items():
 row_ix = where(y == label)[0]
 pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

```
Counter({0: 9990, 1: 10})
```



Se ajusta el modelo correspondiente

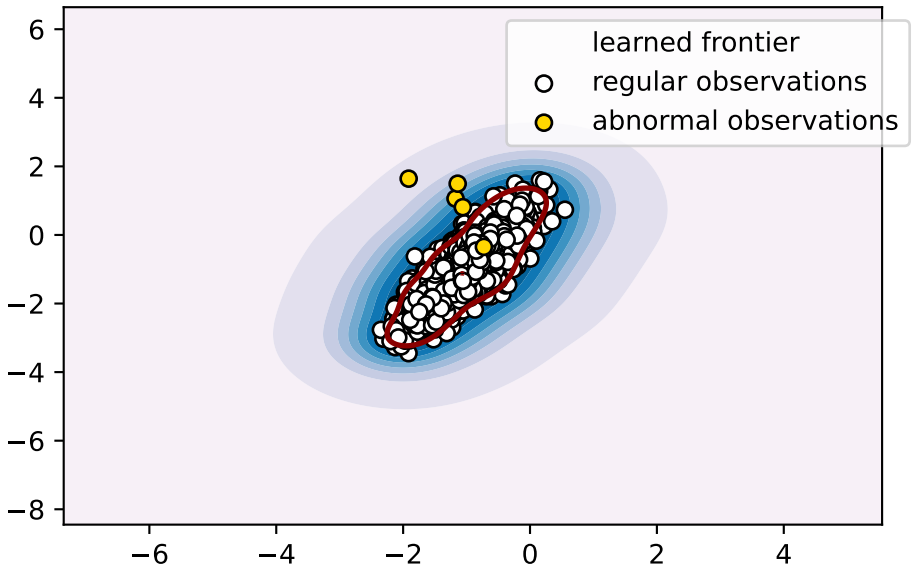
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score
from sklearn.svm import OneClassSVM
# split into train/test sets
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, random_state=2, stratify
# define outlier detection model
model = OneClassSVM(gamma='scale', nu=0.01)
# fit on majority class
trainX_clean = trainX[trainy==0]
model.fit(trainX_clean)
```

```
# detect outliers in the test set
yhat = model.predict(testX)
# mark inliers 1, outliers -1
testy_clean = testy.copy()
testy_clean[testy == 1] = -1
testy_clean[testy == 0] = 1
# calculate score
score = f1_score(testy_clean, yhat, pos_label=-1)
print('F1 Score: %.3f' % score)
```

```
F1 Score: 0.123
```

```
import numpy as np
import matplotlib.pyplot as plt
# define the meshgrid
x_min, x_max = trainX[:, 0].min() - 5, trainX[:, 0].max() + 5
y_min, y_max = trainX[:, 1].min() - 5, trainX[:, 1].max() + 5

x_ = np.linspace(x_min, x_max, 500)
y_ = np.linspace(y_min, y_max, 500)

xx, yy = np.meshgrid(x_, y_)

# evaluate the decision function on the meshgrid
z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])
z = z.reshape(xx.shape)

# plot the decision function and the reduced data
plt.contourf(xx, yy, z, cmap=plt.cm.PuBu)
a = plt.contour(xx, yy, z, levels=[0], linewidths=2, colors='darkred')
b = plt.scatter(trainX[trainy == 0, 0], trainX[trainy == 0, 1], c='white', edgecolors='k')
c = plt.scatter(trainX[trainy == 1, 0], trainX[trainy == 1, 1], c='gold', edgecolors='k')
plt.legend([a.collections[0], b, c], ['learned frontier', 'regular observations', 'abnormal
plt.axis('tight')
plt.show()
```

```
/var/folders/4d/qj4qr8zx1n36td0hlt0p7x_h0000gn/T/ipykernel_24803/4105701295.py:21: Matplotli

The collections attribute was deprecated in Matplotlib 3.8 and will be removed two minor rel
```

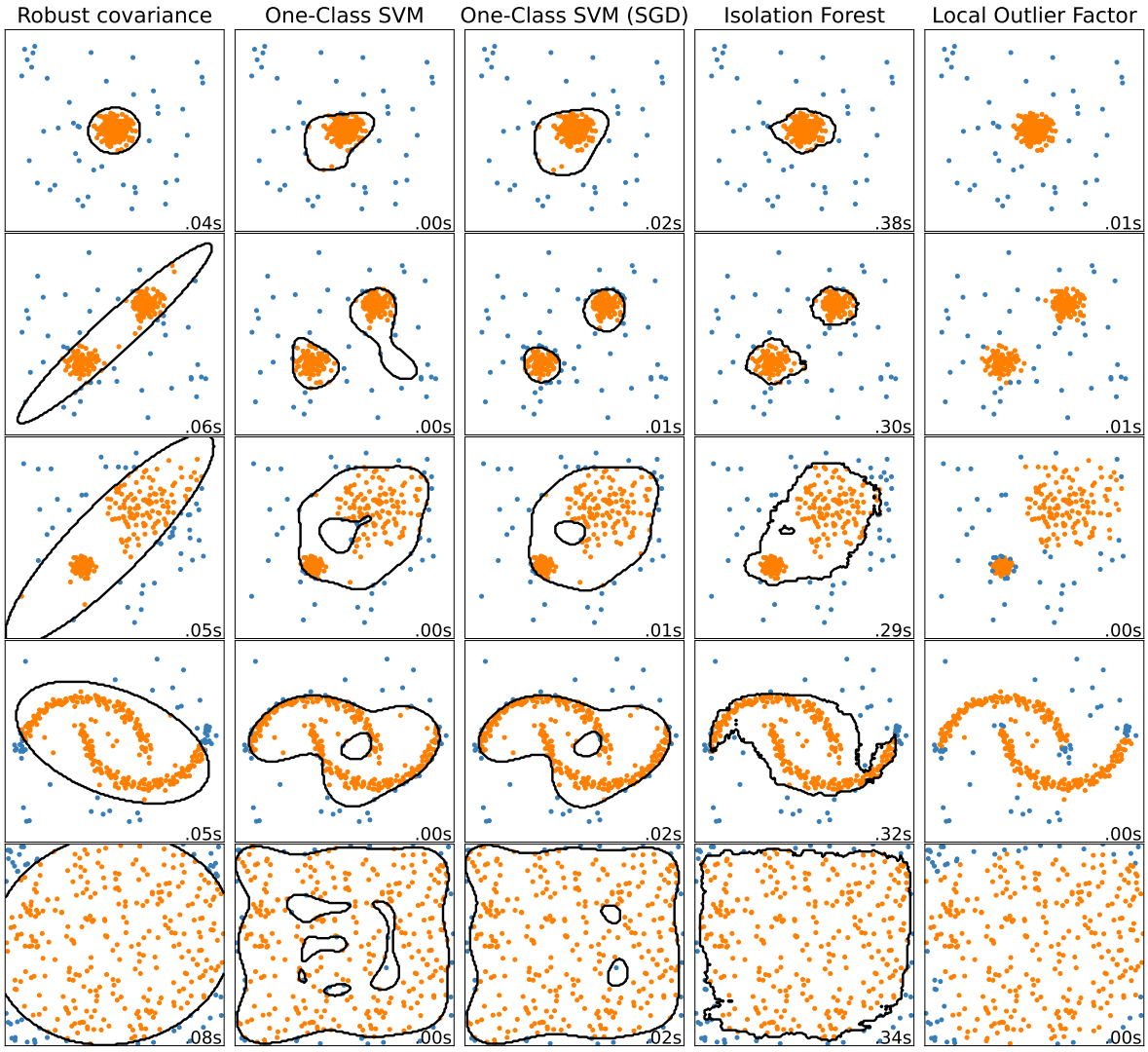



```
import time

import matplotlib
import matplotlib.pyplot as plt
import numpy as np

from sklearn import svm
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs, make_moons
from sklearn.ensemble import IsolationForest
from sklearn.kernel_approximation import Nystroem
from sklearn.linear_model import SGDOneClassSVM
from sklearn.neighbors import LocalOutlierFactor
from sklearn.pipeline import make_pipeline

matplotlib.rcParams["contour.negative_linestyle"] = "solid"

# Example settings
n_samples = 300
outliers_fraction = 0.15
n_outliers = int(outliers_fraction * n_samples)
n_inliers = n_samples - n_outliers

# define outlier/anomaly detection methods to be compared.
# the SGDOneClassSVM must be used in a pipeline with a kernel approximation
```

```
# to give similar results to the OneClassSVM
anomaly_algorithms = [
    (
        "Robust covariance",
        EllipticEnvelope(contamination=outliers_fraction, random_state=42),
    ),
    ("One-Class SVM", svm.OneClassSVM(nu=outliers_fraction, kernel="rbf", gamma=0.1)),
    (
        "One-Class SVM (SGD)",
        make_pipeline(
            Nystroem(gamma=0.1, random_state=42, n_components=150),
            SGDOneClassSVM(
                nu=outliers_fraction,
                shuffle=True,
                fit_intercept=True,
                random_state=42,
                tol=1e-6,
            ),
        ),
    ),
    (
        "Isolation Forest",
        IsolationForest(contamination=outliers_fraction, random_state=42),
    ),
    (
        "Local Outlier Factor",
        LocalOutlierFactor(n_neighbors=35, contamination=outliers_fraction),
    ),
]

# Define datasets
blobs_params = dict(random_state=0, n_samples=n_inliers, n_features=2)
datasets = [
    make_blobs(centers=[[0, 0], [0, 0]], cluster_std=0.5, **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[0.5, 0.5], **blobs_params)[0],
    make_blobs(centers=[[2, 2], [-2, -2]], cluster_std=[1.5, 0.3], **blobs_params)[0],
    4.0
    * (
        make_moons(n_samples=n_samples, noise=0.05, random_state=0)[0]
        - np.array([0.5, 0.25])
    ),
    14.0 * (np.random.RandomState(42).rand(n_samples, 2) - 0.5),
```

```
]

# Compare given classifiers under given settings
xx, yy = np.meshgrid(np.linspace(-7, 7, 150), np.linspace(-7, 7, 150))

plt.figure(figsize=(len(anomaly_algorithms) * 2 + 4, 12.5))
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)

plot_num = 1
rng = np.random.RandomState(42)

for i_dataset, X in enumerate(datasets):
    # Add outliers
    X = np.concatenate([X, rng.uniform(low=-6, high=6, size=(n_outliers, 2))], axis=0)

    for name, algorithm in anomaly_algorithms:
        t0 = time.time()
        algorithm.fit(X)
        t1 = time.time()
        plt.subplot(len(datasets), len(anomaly_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        # fit the data and tag outliers
        if name == "Local Outlier Factor":
            y_pred = algorithm.fit_predict(X)
        else:
            y_pred = algorithm.fit(X).predict(X)

        # plot the levels lines and the points
        if name != "Local Outlier Factor":  # LOF does not implement predict
            Z = algorithm.predict(np.c_[xx.ravel(), yy.ravel()])
            Z = Z.reshape(xx.shape)
            plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors="black")

        colors = np.array(["#377eb8", "#ff7f00"])
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[(y_pred + 1) // 2])

        plt.xlim(-7, 7)
        plt.ylim(-7, 7)
```

```
        plt.xticks(())
        plt.yticks(())
        plt.text(
            0.99,
            0.01,
            ("%.2fs" % (t1 - t0)).lstrip("0"),
            transform=plt.gca().transAxes,
            size=15,
            horizontalalignment="right",
        )
        plot_num += 1

plt.show()
```



13.4 Clasificación multi-etiqueta

En la clasificación multietiqueta, cada ejemplo de entrenamiento no tiene sólo una etiqueta, sino varias. de ellas. Por ejemplo, si queremos describir una imagen, podríamos asignarle varias etiquetas: “gente”, “concierto”, “naturaleza”, o las tres a la vez.

Si el número de valores posibles para las etiquetas es elevado, pero todos son de la misma naturaleza, se puede optar por transformar cada ejemplo etiquetado en varios ejemplos etiquetados, uno por etiqueta. Estos nuevos ejemplos tienen todos el mismo vector de características y una sola etiqueta. Esto se convierte en un problema de clasificación multiclase.Se puede resolver

utilizando la estrategia de uno contra el resto. La única diferencia con el problema multiclase habitual es que se genera un nuevo hiperparámetro: Threshold (umbral).

Si la puntuación de predicción para alguna etiqueta está por encima del umbral, esta etiqueta se predice para el vector de características de entrada. En este escenario, se pueden predecir múltiples etiquetas para un solo vector de características.El valor del umbral se elige utilizando el conjunto de validación. De forma análoga, los algoritmos que pueden convertirse de forma natural en multiclase (árboles de decisión, regresión logística y redes neuronales, entre otros) pueden aplicarse a problemas de clasificación multietiqueta, ya que, estos devuelven la puntuación de cada clase, entonces se puede definir un umbral y asignar varias etiquetas a un vector de características si el umbral está por encima de un valor elegido, usando de forma experimental el conjunto de validación.

Los algoritmos de redes neuronales pueden entrenar de forma natural modelos de clasificación multietiqueta utilizando la función de costo de entropía cruzada binaria. La capa de salida de la red neuronal, en este caso,tiene una unidad por etiqueta. Cada unidad de la capa de salida tiene la función de activación sigmoidea.

En los casos en los que el número de posibles valores que puede tomar cada etiqueta es pequeño, se puede converti en un problema multiclase utilizando un enfoque diferente. Imaginemos el siguiente problema. Se quiere etiquetar imágenes y las etiquetas pueden ser de dos tipos. El primer tipo de etiqueta puede tener dos valores posibles: {foto, pintura}; la etiqueta del segundo tipo puede tener tres valores posibles valores {retrato, paisaje, otro}. Se puede crear una nueva clase falsa para cada combinación de las dos clases originales, así:

Tabla

Clase falsa

Clase real 1

Clase real 2

1

Foto

Retrato

2

Foto

Paisaje

3

Foto

Otro

4

Pintura

Retrato

5

Pintura

Paisaje

6

Pintura

Otro

Ahora se tienen los mismos ejemplos etiquetados, se sustituyen las etiquetas múltiples reales por una etiqueta falsa con valores de 1 a 6. Este enfoque funciona bien en la práctica cuando no hay demasiadas combinaciones posibles de clases. De lo contrario, es necesario utilizar muchos más datos de entrenamiento para compensar el aumento del número de clases.

La principal ventaja de este enfoque es que mantiene correlacionadas las etiquetas, al contrario que los métodos vistos anteriormente que predicen cada etiqueta independientemente de la otra. La correlación entre etiquetas puede ser una propiedad esencial en muchos problemas. Por ejemplo, si quiere predecir si un mensaje de correo electrónico es spam o no_spam al mismo tiempo que como predecir si es correo ordinario o prioritario.

13.4.1 Ejemplo código

13.4.1.1 Formato de destino

Una representación válida de multi etiqueta es una matriz binaria y de forma densa o escasa . Cada columna representa una clase. Los 1' en cada fila indican las clases positivas con las que se ha etiquetado una muestra. Un ejemplo de matriz densa para 3 muestras:(n_samples, n_classes)

```
import numpy as np
y = np.array([[1, 0, 0, 1], [0, 0, 1, 1], [0, 0, 0, 0]])
print(y)
```

```
[[1 0 0 1]
 [0 0 1 1]
 [0 0 0 0]]
```

También se pueden crear matrices densas utilizando MultiLabelBinarizer

```
import numpy as np
import scipy.sparse as sparse

y = np.array([[1, 0, 0, 1], [0, 0, 1, 1], [0, 0, 0, 0]])
y_sparse = sparse.csr_matrix(y)
print(y_sparse)
```

```
  (0, 0)    1
  (0, 3)    1
  (1, 2)    1
  (1, 3)    1
```

```
from sklearn.datasets import make_multilabel_classification
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, hamming_loss

# Generate synthetic multi-label dataset
X, y = make_multilabel_classification(n_samples=100, n_features=10, n_classes=5, random_state

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a multi-label classifier
classifier = MultiOutputClassifier(KNeighborsClassifier())

# Train the classifier
classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = classifier.predict(X_test)

# Calculate accuracy and Hamming loss
accuracy = accuracy_score(y_test, y_pred)
hamming_loss = hamming_loss(y_test, y_pred)

print("Hamming Loss:", hamming_loss)
```

```
Hamming Loss: 0.21
```

14 Práctica avanzada

14.1 Manejo de datos desbalanceados

El desbalance de clases en los conjuntos de datos es un desafío común en el aprendizaje automático, particularmente en aplicaciones como la detección de fraude, donde las clases de interés suelen estar subrepresentadas. Este desbalance puede sesgar el rendimiento de los modelos hacia la clase mayoritaria, resultando en una pobre clasificación de las instancias de la clase minoritaria.

14.1.1 Solución mediante Pesos Diferenciales en SVM

El SVM (Support Vector Machine) con margen blando permite manejar el desbalance mediante la asignación de un costo diferente a las clasificaciones erróneas de las clases. Matemáticamente, esto se refleja en la función de pérdida, donde el costo CC se ajusta por clase:

$$L(y, f(x)) = C_{clase} max(0, 1 y f(x)) 2$$

donde C_{clase} es el peso asignado a la clase, y es la etiqueta verdadera, y $f(x)$ es la decisión del modelo.

14.1.1.1 Implementación

Veamos cómo implementar un clasificador SVM que gestiona el desbalance de clases asignando pesos específicos en scikit-learn.

Primero, creemos un conjunto de datos desbalanceado y dividámoslo en conjuntos de entrenamiento y prueba.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# Generamos un conjunto de datos desbalanceado
```

```
X, y = make_classification(n_classes=2, class_sep=2,
                           weights=[0.1, 0.90], n_informative=3, n_redundant=1, flip_y=0,
                           n_features=20, n_clusters_per_class=1, n_samples=1000, random_sta

# Dividimos en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
```

Ahora, ajustaremos un modelo SVM considerando el desbalance mediante el uso de pesos de clase:

```
# Definimos los pesos de las clases para tratar el desbalance
weights = {0: 1000, 1: 1}  # Aumentamos el peso de la clase minoritaria

# Creamos el clasificador SVM con los pesos de clase
clf = SVC(kernel='linear', class_weight=weights)

# Entrenamos el modelo
clf.fit(X_train, y_train)

# Evaluamos el modelo
accuracy = clf.score(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.98
```

14.1.2 Solución #2: Submuestreo y Sobremuestreo

Si el algoritmo de aprendizaje no permite la ponderación de clases, existen técnicas de muestreo como el sobremuestreo (oversampling), el submuestreo (undersampling), y la creación de ejemplos sintéticos mediante algoritmos como SMOTE o ADASYN para equilibrar las clases.

SMOTE (Synthetic Minority Over-sampling Technique) es una técnica de sobremuestreo que crea ejemplos sintéticos de la clase minoritaria para equilibrar el conjunto de datos. Veamos cómo aplicarlo usando la biblioteca `imbalanced-learn`.

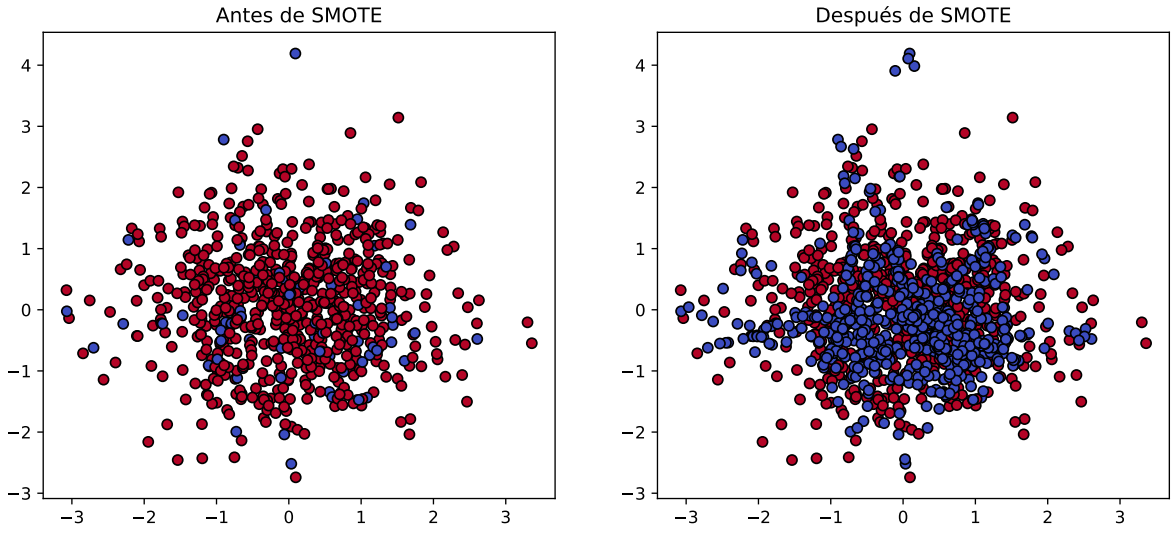
```
import matplotlib.pyplot as plt
from imblearn.over_sampling import SMOTE
from sklearn.metrics import classification_report

# Aplicamos SMOTE al conjunto de entrenamiento
smote = SMOTE(random_state=42)
```

```
X_res, y_res = smote.fit_resample(X_train, y_train)

# Entrenamos un nuevo clasificador SVM con los datos sobremuestreados
clf_smote = SVC(kernel='linear')
clf_smote.fit(X_res, y_res)

# Evaluamos el modelo
y_pred = clf_smote.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.96      0.93        26
           1       1.00      0.99      0.99       224

    accuracy                           0.98       250
   macro avg       0.94      0.97      0.96       250
weighted avg       0.98      0.98      0.98       250
```

Comparamos el efecto antes y después de aplicar SMOTE:

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.coolwarm, edgecolors='k')
plt.title('Antes de SMOTE')

plt.subplot(1, 2, 2)
plt.scatter(X_res[:, 0], X_res[:, 1], c=y_res, cmap=plt.cm.coolwarm, edgecolors='k')
plt.title('Después de SMOTE')

plt.show()
```



14.2 Combinación de Modelos

La combinación de modelos en el aprendizaje automático es una técnica poderosa que busca mejorar el rendimiento predictivo al integrar las fortalezas de varios modelos. Existen diversas formas de combinar modelos, siendo las más comunes el promedio (averaging), el voto de mayoría (majority vote) y el apilamiento (stacking). Cada uno de estos métodos tiene aplicaciones específicas y beneficios únicos.

14.2.1 Promedio (Averaging)

El método de promedio es aplicable tanto para la regresión como para la clasificación. Consiste en aplicar todos los modelos base al input x y luego promediar las predicciones. En clasificación, se promedian las probabilidades predichas para cada clase.

Consideremos un conjunto de datos de regresión y combinemos las predicciones de varios modelos de regresión mediante el promedio.

```
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.linear_model import LinearRegression
import numpy as np

# Generamos un conjunto de datos de regresión
X, y = make_regression(n_samples=1000, n_features=20, noise=0.1, random_state=42)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Entrenamos varios modelos
model_rf = RandomForestRegressor(n_estimators=10, random_state=42).fit(X_train, y_train)
model_gb = GradientBoostingRegressor(n_estimators=10, random_state=42).fit(X_train, y_train)
model_lr = LinearRegression().fit(X_train, y_train)

# Predecimos y promediamos las predicciones
predictions = np.mean([model_rf.predict(X_test), model_gb.predict(X_test), model_lr.predict(X

# Evaluamos el rendimiento del modelo promedio
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_test, predictions)
mse_rf = mean_squared_error(y_test, model_rf.predict(X_test))
mse_gb = mean_squared_error(y_test, model_gb.predict(X_test))
mse_lr = mean_squared_error(y_test, model_lr.predict(X_test))

print(f'MSE del modelo RF: {mse_rf}')
print(f'MSE del modelo GB: {mse_gb}')
print(f'MSE del modelo LR: {mse_lr}')
print(f'MSE del modelo promediado: {mse}')
```

```
MSE del modelo RF: 9445.970722326663
MSE del modelo GB: 20390.298586678025
MSE del modelo LR: 0.010704979443048707
MSE del modelo promediado: 5842.696381684256
```

14.2.2 Voto de mayoría (Majority Vote)

El voto de mayoría se utiliza para modelos de clasificación. Se aplica cada uno de los modelos base al input xx y se selecciona la clase que obtenga la mayoría de votos entre todas las predicciones.

Utilizaremos varios clasificadores y combinaremos sus predicciones mediante el voto de mayoría.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

```
from sklearn.metrics import classification_report

# Generamos un conjunto de datos de clasificación
X, y = make_classification(n_samples=1000, n_features=20, n_classes=2, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Entrenamos varios clasificadores
model_rf = RandomForestClassifier(n_estimators=10, random_state=42)
model_lr = LogisticRegression()
model_svc = SVC(probability=True, random_state=42)

# Combinamos mediante voto de mayoría
eclf = VotingClassifier(estimators=[('rf', model_rf), ('lr', model_lr), ('svc', model_svc)],
eclf.fit(X_train, y_train)
model_rf.fit(X_train, y_train)
model_lr.fit(X_train, y_train)
model_svc.fit(X_train, y_train)


# Evaluamos el rendimiento
accuracy = eclf.score(X_test, y_test)
accuracy_rf = model_rf.score(X_test, y_test)
accuracy_lr = model_lr.score(X_test, y_test)
accuracy_svc = model_svc.score(X_test, y_test)

print(f'Accuracy del modelo RF: {accuracy_rf}')
print(f'Accuracy del modelo LR: {accuracy_lr}')
print(f'Accuracy del modelo SVC: {accuracy_svc}')
print(f'Accuracy del modelo combinado mediante voto de mayoría: {accuracy}')

y_pred = eclf.predict(X_test)
print(classification_report(y_test, y_pred))

print(classification_report(y_test, model_rf.predict(X_test)))
```

```
Accuracy del modelo RF: 0.8333333333333334
Accuracy del modelo LR: 0.85
Accuracy del modelo SVC: 0.8333333333333334
Accuracy del modelo combinado mediante voto de mayoría: 0.8433333333333334
              precision    recall  f1-score   support

           0       0.81      0.88      0.84       145
```

```
           1       0.88      0.81      0.84       155

    accuracy                           0.84       300
   macro avg       0.84      0.84      0.84       300
weighted avg       0.85      0.84      0.84       300

              precision    recall  f1-score   support

           0       0.80      0.88      0.84       145
           1       0.88      0.79      0.83       155

    accuracy                           0.83       300
   macro avg       0.84      0.83      0.83       300
weighted avg       0.84      0.83      0.83       300
```

14.2.3 Apilamiento (Stacking)

El apilamiento consiste en combinar varios modelos base y utilizar sus salidas como input para un meta-modelo, que hace la predicción final. Ejemplo en Python para Stacking

Implementaremos stacking con varios modelos base y un meta-modelo de regresión logística.

```python
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

# Definimos los modelos base y el meta-modelo
base_models = [('rf', RandomForestClassifier(n_estimators=10, random_state=42)),
               ('svc', SVC(probability=True, random_state=42))]
meta_model = LogisticRegression()

# Creamos el modelo de apilamiento
stacking_model = StackingClassifier(estimators=base_models, final_estimator=meta_model)

# Entrenamos y evaluamos el modelo de apilamiento
stacking_model.fit(X_train, y_train)

accuracy = stacking_model.score(X_test, y_test)
print(f'Accuracy del modelo de apilamiento: {accuracy}')

print(classification_report(y_test, stacking_model.predict(X_test)))
print(classification_report(y_test, model_rf.predict(X_test)))
print(classification_report(y_test, model_svc.predict(X_test)))
```

```
Accuracy del modelo de apilamiento: 0.84
              precision    recall  f1-score   support

           0       0.82      0.86      0.84       145
           1       0.86      0.82      0.84       155

    accuracy                           0.84       300
   macro avg       0.84      0.84      0.84       300
weighted avg       0.84      0.84      0.84       300

              precision    recall  f1-score   support

           0       0.80      0.88      0.84       145
           1       0.88      0.79      0.83       155

    accuracy                           0.83       300
   macro avg       0.84      0.83      0.83       300
weighted avg       0.84      0.83      0.83       300

              precision    recall  f1-score   support

           0       0.81      0.86      0.83       145
           1       0.86      0.81      0.83       155

    accuracy                           0.83       300
   macro avg       0.83      0.83      0.83       300
weighted avg       0.83      0.83      0.83       300
```

14.3 Entrenamiento de Redes Neuronales

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Crear un generador de datos con normalización
datagen = ImageDataGenerator(rescale=1./255)

# Suponiendo que 'directorio_de_datos' es el camino a las imágenes
train_generator = datagen.flow_from_directory(
    directorio_de_datos,
```

```
    target_size=(200, 200),  # Todas las imágenes se redimensionan a 200x200
    batch_size=32,
    class_mode='binary'  # o 'categorical' para clasificación multiclase
)

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Crear un tokenizador para convertir palabras a índices
tokenizer = Tokenizer(num_words=10000)  # Considera las 10,000 palabras más comunes
tokenizer.fit_on_texts(textos)  # 'textos' es una lista de documentos de texto

# Convertir textos en secuencias de índices
sequences = tokenizer.texts_to_sequences(textos)

# Acolchar secuencias para que tengan la misma longitud
data = pad_sequences(sequences, maxlen=100)  # Longitud fija de 100 para todas las secuencias

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Crear un modelo simple como punto de partida
model = Sequential([
    Dense(64, activation='relu', input_shape=(100,)),  # Ejemplo para datos vectorizados de l
    Dropout(0.5),  # Regularización mediante Dropout
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Entrenar el modelo
model.fit(data, etiquetas, epochs=10, validation_split=0.2)  # 'etiquetas' es un array de eti
```

15 Aprendizaje no supervisado

15.1 Estimación de densidad

La estimación de la densidad es un problema de modelar la función de densidad (*pdf*) de la distribución desconcida del dataset. Sus aplicaciones principales son en la ditección de novedades e intrusiones. Anteriormente se trabajó con la estimación de la *pdf* para el caso paramétrico con la distribución normal multivariada. Acá usaremos el método del kernel, ya que este es no paramétrico.

Así, sea $\{x_i\}_{i=1}^N$ un dataset de una dimensión donde las muestras son construidas a partir de una *pdf* desconocida f con $x_i \in \mathbb{R},\ \forall i = 1, \ldots, N$. Estamos interesados en modelar la curva de la función f. Con nuestro modelo de kernel, denotado por $\hat{f}$, defindo como:

$$\hat{f}_h(x) = \frac{1}{Nh} \sum_{i=1}^{N} k\left(\frac{x - x_i}{h}\right) \tag{15.1}$$

Donde h es un hiperparámetro que controla la relación sesgo-varianza. Acá usaremos el kernel gaussiano:

$$k(z) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-z^2}{2}\right) \tag{15.2}$$

Nosotros buscamos el valor de h que minimiza la diferencia entre la curva original f y la curva aproximada de nuestro modelo f_h. Una medida razonable para esta diferencia es el error cuadrático medio integrado (MISE, por sus siglas en inglés), definido por:

$$MISE(b) = \mathbb{E}\left[\int_{\mathbb{R}} \left(\hat{f}_h(x) - f(x)\right)^2 dx\right] \tag{15.3}$$

En la ecuación (Ecuación **??**) la integral $\int_{\mathbb{R}}$ remplaza a la sumatoria $\sum_{i=1}^{N}$ que empleamos en el promedio, mientras que la esperanza $\mathbb{E}$ reemplaza el promedio $\frac{1}{N}$.

Notese que cuando la función de pérdida es continua como la función de $\left(\hat{f}_h(x) - f(x)\right)^2$, se reemplaza la sumatoria por la integrasl. El operador de esperanza $\mathbb{E}$ siginifica que queremos que h sea el óptimo para todos las posibilidades del set de entrenamiento. Esto es importante debido

a que $\hat{f}_h$ es definido en un conjunto finito de datos de alguna distribución de probabilidad; mientras que la *pdf* real f está definida en un dominio infinito $\mathbb{R}$.

Note que, reescribiendo el lado derecho de la (Ecuación **??**), obtenemos

$$\mathbb{E}\left[\int_{\mathbb{R}} \hat{f}_h^2(x)dx\right] - 2\mathbb{E}\left[\int_{\mathbb{R}} \hat{f}_h(x)f(x)dx\right] + \mathbb{E}\left[\int_{\mathbb{R}} f^2(x)dx\right]$$

Note que el tercer término es independiente de h y podría ser ignorado. Un estimador insesgado del primer término está dado por $\int_{\mathbb{R}} \hat{f}_b^2(x)dx$, mientras que el estimador insesgado para el segundo término está aproximado por $\frac{-2}{N}\sum_{i=1}^{N} \hat{f}_h^{(i)}(x_i)$, donde $\hat{f}_h^{(i)}(x_i)$ es el kernel con los datos de entrenamiento menos el dato x_i.

El término $\sum_{i=1}^{N} \hat{f}_h^{(i)}(x_i)$ es conocindo como el estimador de dejar una estimación por fuera (*leave one out estimate*); es una forma de validación cruzada donde cada *fold* contienen una muestra. Además, se puede ver como $\int_{\mathbb{R}} \hat{f}_h(x)f(x)dx$ es la esperanza de la función $\hat{f}_h$, esto por que f es una función de densidad. Se puede demostra que el estimador *leave one out estimate* es un estimador insesgado para $\mathbb{E}\left[\int_{\mathbb{R}} \hat{f}_h(x)f(x)dx\right]$.

Ahora, para hallar el valor óptimo h^* para h, queremos minimizar la función de costo definida por:

$$\int_{\mathbb{R}} \hat{f} * h^2(x)dx - \frac{2}{N}\sum *i = 1^N \hat{f}_h^{(i)}(x_i)$$

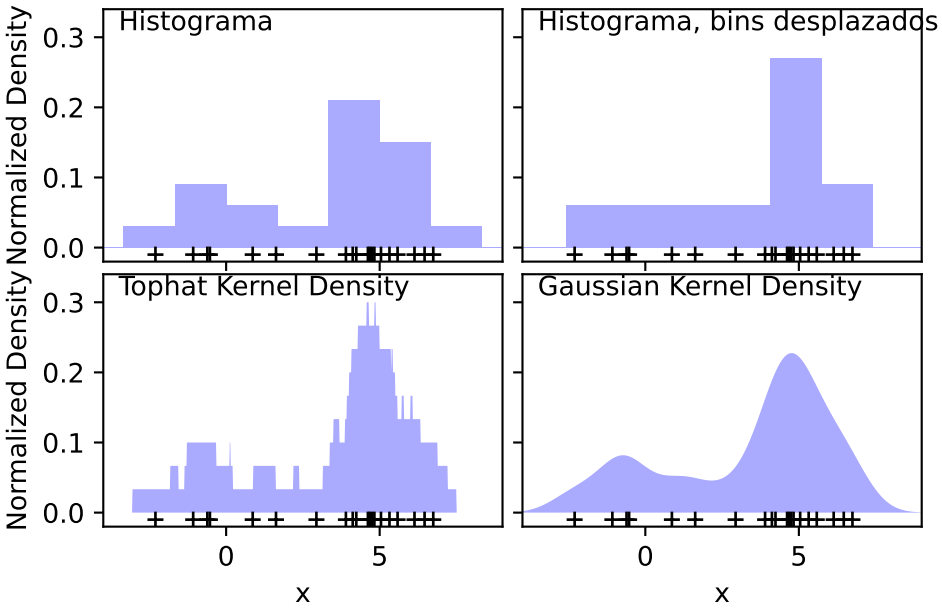
Se puede hallar h^* utilizando *grid search*¨. Para D dimensiones, el término del error $x - x_i$ de la (Ecuación **??**) puede ser reemplazado por la norma euclidea $||\curvearrowleft - \curvearrowleft_i||$.

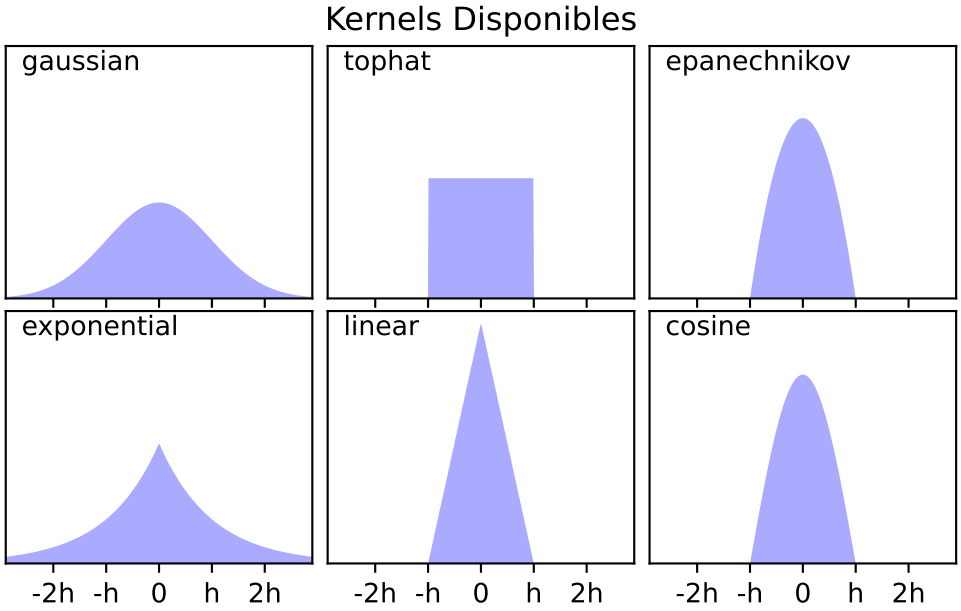
```
#Importar las librerías
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm
from sklearn.neighbors import KernelDensity

# ----------------------------------------------------------------------
# Plot the progression of histograms to kernels
np.random.seed(1)
N = 20
X = np.concatenate(
    (np.random.normal(0, 1, int(0.3 * N)), np.random.normal(5, 1, int(0.7 * N)))
)[:, np.newaxis]
X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]
bins = np.linspace(-5, 10, 10)
```

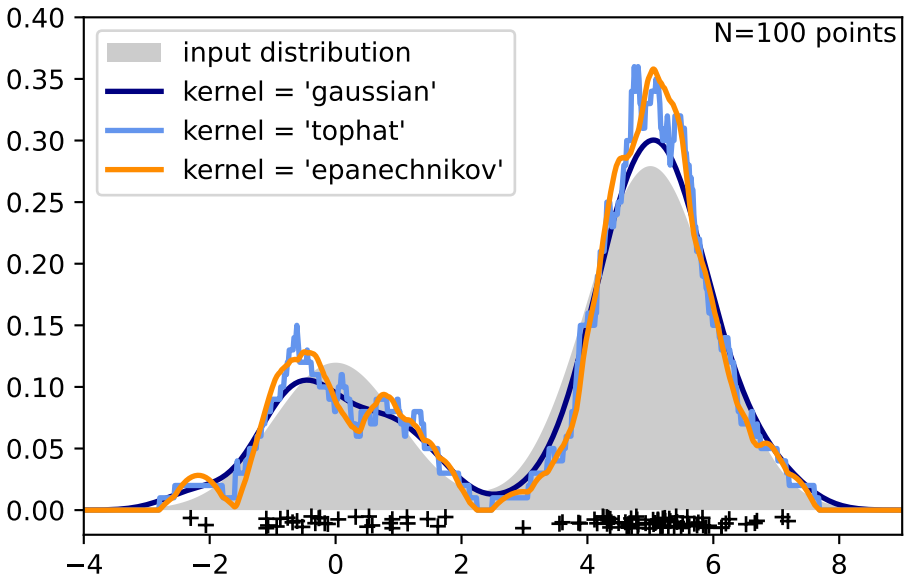
```
fig, ax = plt.subplots(2, 2, sharex=True, sharey=True)
fig.subplots_adjust(hspace=0.05, wspace=0.05)

# Histograma
ax[0, 0].hist(X[:, 0], bins=bins, fc="#AAAAFF", density=True)
ax[0, 0].text(-3.5, 0.31, "Histograma")

# Histograma con las particiones desplazadas
ax[0, 1].hist(X[:, 0], bins=bins + 0.75, fc="#AAAAFF", density=True)
ax[0, 1].text(-3.5, 0.31, "Histograma, bins desplazados")

# tophat KDE
kde = KernelDensity(kernel="tophat", bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 0].fill(X_plot[:, 0], np.exp(log_dens), fc="#AAAAFF")
ax[1, 0].text(-3.5, 0.31, "Tophat Kernel Density")

# Gaussian KDE
kde = KernelDensity(kernel="gaussian", bandwidth=0.75).fit(X)
log_dens = kde.score_samples(X_plot)
ax[1, 1].fill(X_plot[:, 0], np.exp(log_dens), fc="#AAAAFF")
ax[1, 1].text(-3.5, 0.31, "Gaussian Kernel Density")

for axi in ax.ravel():
    axi.plot(X[:, 0], np.full(X.shape[0], -0.01), "+k")
    axi.set_xlim(-4, 9)
    axi.set_ylim(-0.02, 0.34)

for axi in ax[:, 0]:
    axi.set_ylabel("Normalized Density")

for axi in ax[1, :]:
    axi.set_xlabel("x")

# ----------------------------------------------------------------------
# Plot all available kernels
X_plot = np.linspace(-6, 6, 1000)[:, None]
X_src = np.zeros((1, 1))

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True)
fig.subplots_adjust(left=0.05, right=0.95, hspace=0.05, wspace=0.05)
```

```
def format_func(x, loc):
    if x == 0:
        return "0"
    elif x == 1:
        return "h"
    elif x == -1:
        return "-h"
    else:
        return "%ih" % x


for i, kernel in enumerate(
    ["gaussian", "tophat", "epanechnikov", "exponential", "linear", "cosine"]
):
    axi = ax.ravel()[i]
    log_dens = KernelDensity(kernel=kernel).fit(X_src).score_samples(X_plot)
    axi.fill(X_plot[:, 0], np.exp(log_dens), "-k", fc="#AAAAFF")
    axi.text(-2.6, 0.95, kernel)

    axi.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
    axi.xaxis.set_major_locator(plt.MultipleLocator(1))
    axi.yaxis.set_major_locator(plt.NullLocator())

    axi.set_ylim(0, 1.05)
    axi.set_xlim(-2.9, 2.9)

ax[0, 1].set_title("Kernels Disponibles")

# ----------------------------------------------------------------------
# Plot a 1D density example
N = 100
np.random.seed(1)
X = np.concatenate(
    (np.random.normal(0, 1, int(0.3 * N)), np.random.normal(5, 1, int(0.7 * N)))
)[:, np.newaxis]

X_plot = np.linspace(-5, 10, 1000)[:, np.newaxis]

true_dens = 0.3 * norm(0, 1).pdf(X_plot[:, 0]) + 0.7 * norm(5, 1).pdf(X_plot[:, 0])

fig, ax = plt.subplots()
ax.fill(X_plot[:, 0], true_dens, fc="black", alpha=0.2, label="input distribution")
```

```
colors = ["navy", "cornflowerblue", "darkorange"]
kernels = ["gaussian", "tophat", "epanechnikov"]
lw = 2

for color, kernel in zip(colors, kernels):
    kde = KernelDensity(kernel=kernel, bandwidth=0.5).fit(X)
    log_dens = kde.score_samples(X_plot)
    ax.plot(
        X_plot[:, 0],
        np.exp(log_dens),
        color=color,
        lw=lw,
        linestyle="-",
        label="kernel = '{0}'".format(kernel),
    )

ax.text(6, 0.38, "N={0} points".format(N))

ax.legend(loc="upper left")
ax.plot(X[:, 0], -0.005 - 0.01 * np.random.random(X.shape[0]), "+k")

ax.set_xlim(-4, 9)
ax.set_ylim(-0.02, 0.4)
plt.show()
```


Histograma
Histograma, bins desplazados
Tophat Kernel Density
Gaussian Kernel Density
Normalized Density
0.3
0.2
0.1
0.0
0
5
x


Kernels Disponibles
gaussian
tophat
epanechnikov
exponential
linear
cosine
-2h
-h
0
h
2h



La agrupación es un problema de aprender a asignar una etiqueta a ejemplos aprovechando un no etiquetado conjunto de datos. Debido a que el conjunto de datos no está etiquetado en absoluto, decidir si el modelo aprendido es óptimo es mucho más complicado que en el aprendizaje supervisado. Existe una variedad de algoritmos de agrupamiento y, desafortunadamente, es difícil saber cuál es el mejor calidad para su conjunto de datos. Generalmente, el rendimiento de cada algoritmo depende de las propiedades desconocidas de la distribución de probabilidad de la que se extrajo el conjunto de datos.

15.2 K-Medias

El algoritmo de agrupamiento de k-medias funciona de la siguiente manera:

Primero, el analista tiene que elegir k - el número de clases (o grupos). Luego colocamos aleatoriamente k vectores de características, llamados centroides, en el espacio de características.

Luego calculamos la distancia desde cada ejemplo x a cada centroide usando alguna métrica, como la distancia euclidiana. Luego asignamos el centroide más cercano a cada ejemplo (como si etiquetáramos cada ejemplo con una identificación de centroide como etiqueta). Para cada centroide, calculamos el vector de características promedio de los ejemplos etiquetados con él. Estas características promedio los vectores se convierten en las nuevas ubicaciones de los centroides.

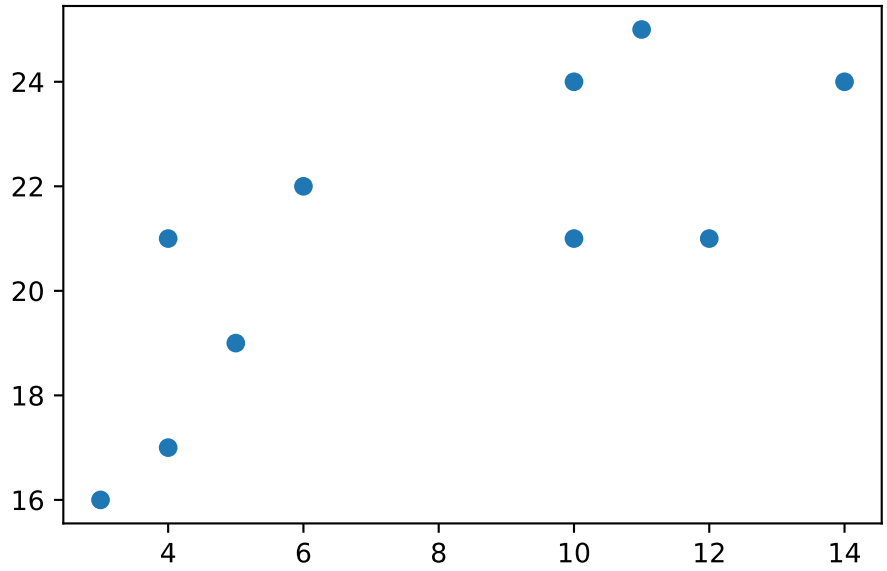
El valor de k, el número de clusters, es un hiperparámetro que los datos deben ajustar. Existen algunas técnicas para seleccionar k. Ninguno de ellos ha demostrado ser óptimo. La mayoría de requieren que el analista haga una “suposición fundamentada” observando algunas

métricas o examinando visualmente las asignaciones de grupos. Más adelante en este capítulo, consideraremos una técnica lo que permite elegir un valor razonablemente bueno para k sin mirar los datos y hacer suposiciones.

15.2.1 Ejemplo

Visualización de datos

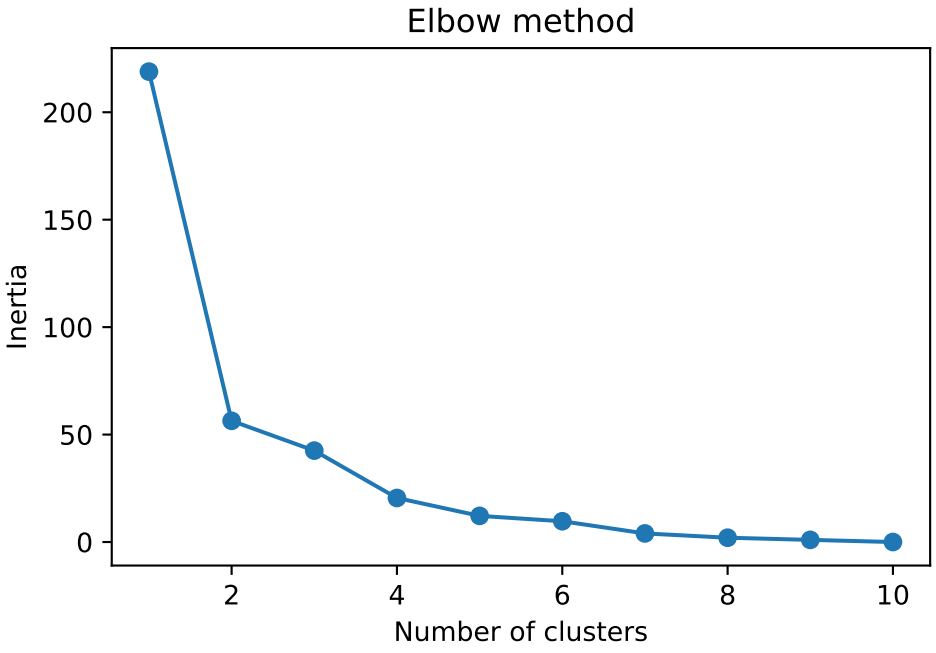
```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()
```



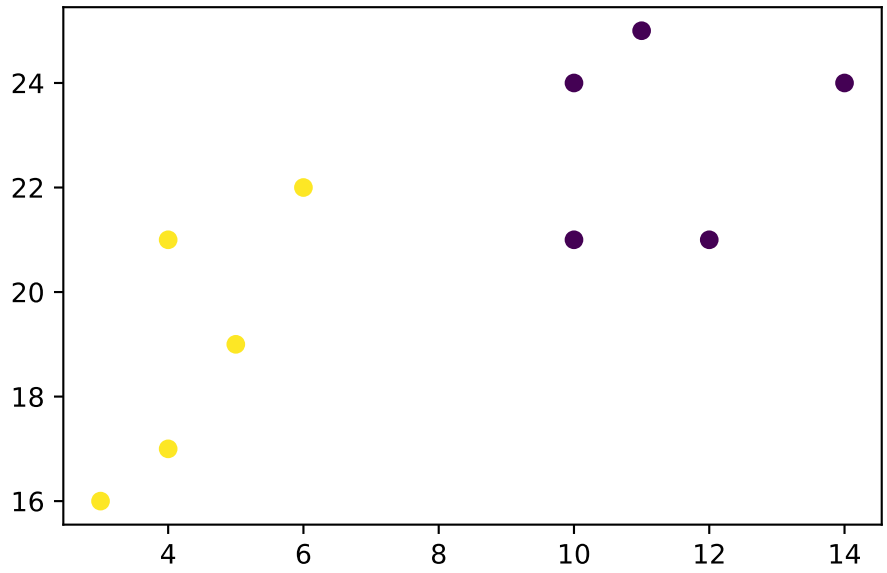
Método del codo para seleccionar la cantidad de k:

```
from sklearn.cluster import KMeans

data = list(zip(x, y))
inertias = []

for i in range(1,11):
```

```
    kmeans = KMeans(n_clusters=i, n_init="auto")
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)

plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```




```
kmeans = KMeans(n_clusters=2, n_init="auto")
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```



15.3 Reducción de dimensionalidad

15.3.1 Análisis de componentes principales (PCA)

15.3.2 UMAP

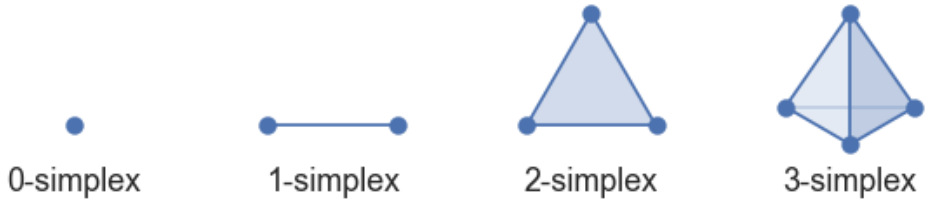
UMAP (Uniform Manifold Approximation and Projection) es un algoritmo de aprendizaje de manifolds para la reducción de dimensionalidad, superior a t-SNE por su eficiencia y versatilidad. Desarrollado por Leland McInnes, John Healy y James Melville en 2018, UMAP se fundamenta en el análisis topológico de datos, ofreciendo una metodología robusta para visualizar y analizar datos en alta dimensión.

El algoritmo construye representaciones topológicas de los datos mediante aproximaciones locales del manifold y uniendo estas representaciones en un conjunto simplicial difuso. Minimiza la entropía cruzada entre las representaciones topológicas de los espacios de alta y baja dimensión para lograr una proyección coherente.

Se dará un resumen básico del método, sin embargo se recomienda leer el artículo original de UMAP para una comprensión más profunda.

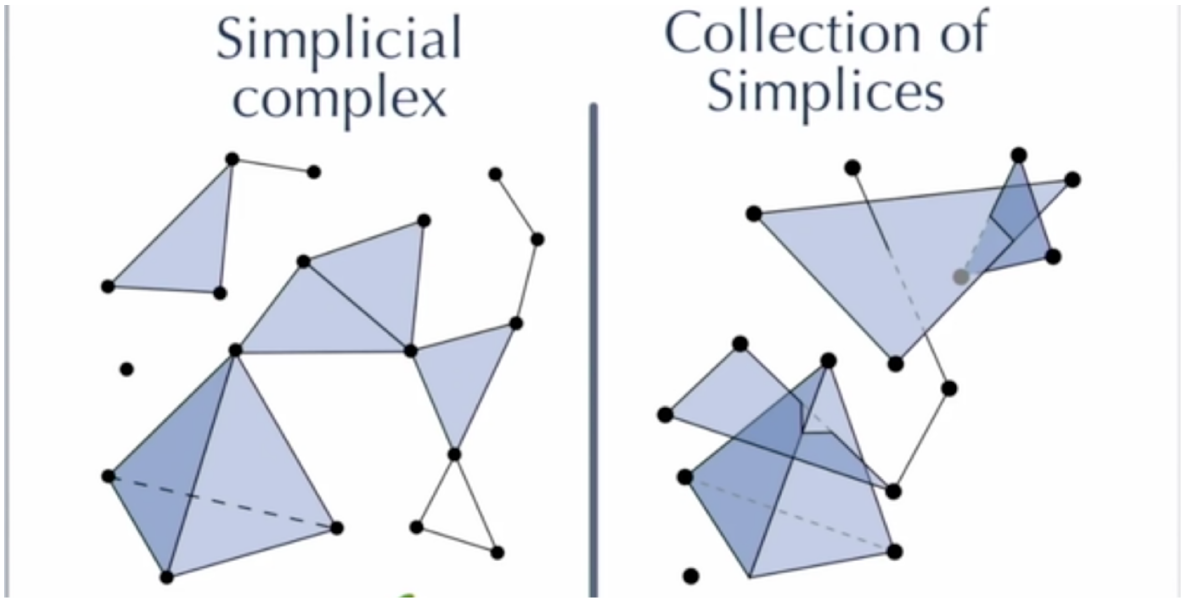
15.3.3 Análisis topológico de datos y complejos simpliciales

Geometricamente, un k-simplex es un objeto k-dimensional que es simplemente la envolutra convexa de $k+1$ puntos en un espacio k-dimensional. Un 0-simplex es un vértice, un 1-simplex es una arista, un 2-simplex es un triángulo, un 3-simplex es un tetraedro, etc.



Un complejo simplicial K es una colección de simplexes que cumple con dos propiedades:

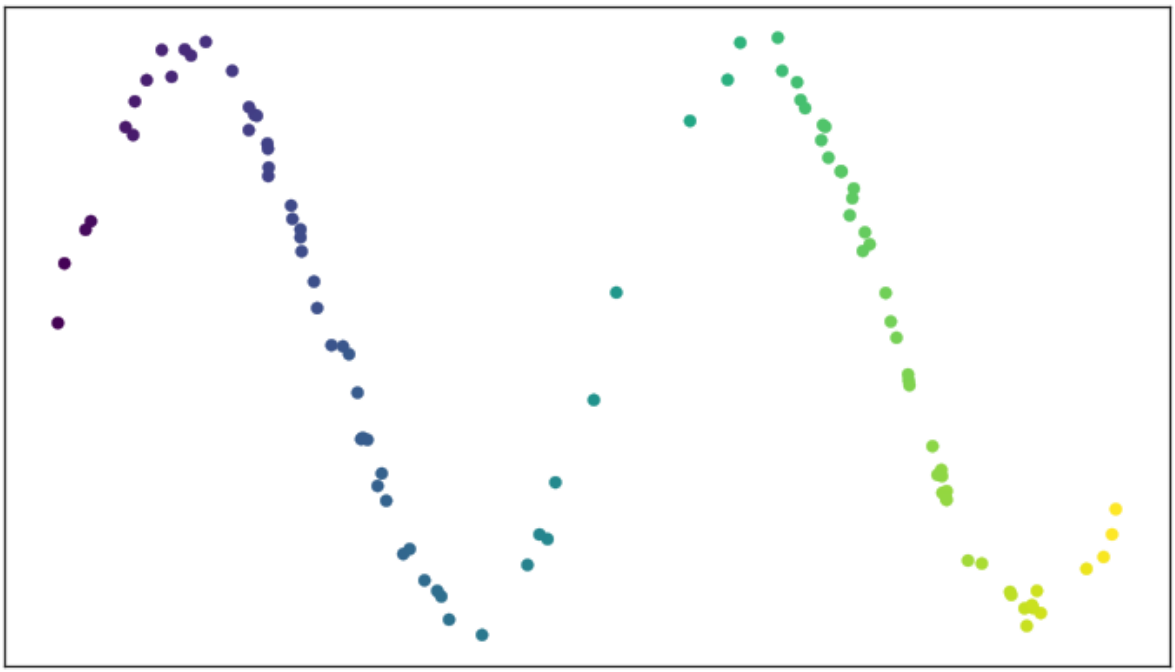
1. Cada cara de un simplex en K también está en K.
2. La intersección de dos simplexes en $\sigma_1, \sigma_2 \in K$ es una cara de ambos σ_1 y σ_2.



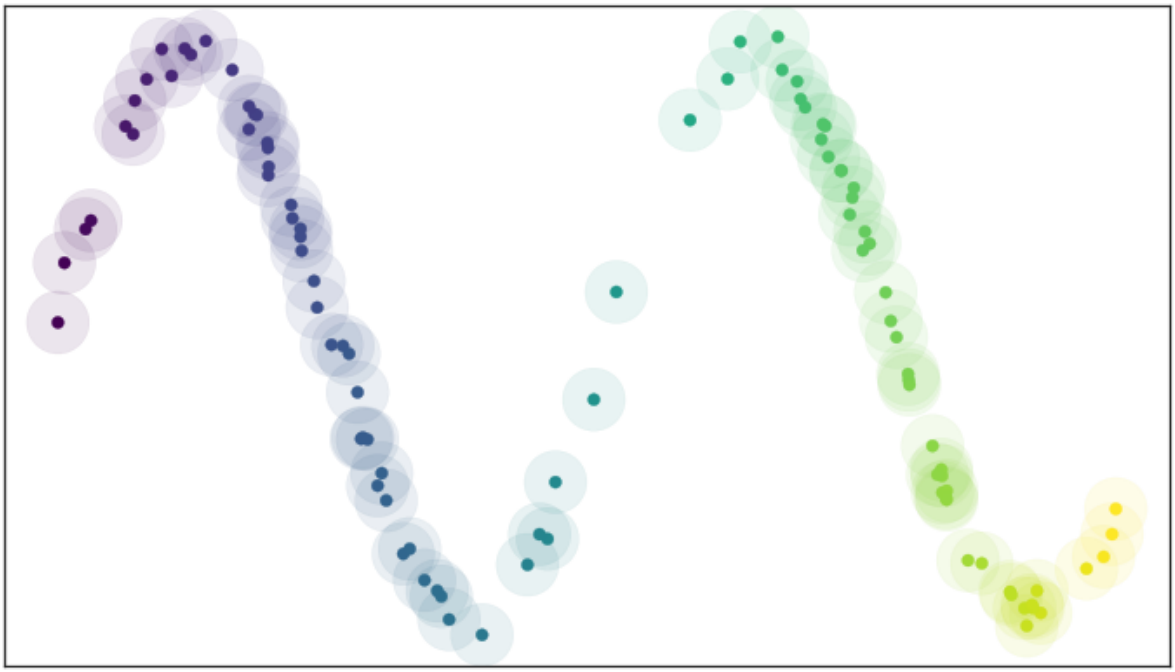
15.3.4 Construcción intuitiva de UMAP

Un conjunto de datos es solo una colección finita de puntos en un espacio. En general para entender las características topoloficas, necesitas crear una covertura abierta del espacio. Si los datos están en un espacio métrico, una forma de aproximar esas coverturas abiertas es con bolas abiertas alrededor de cada punto.

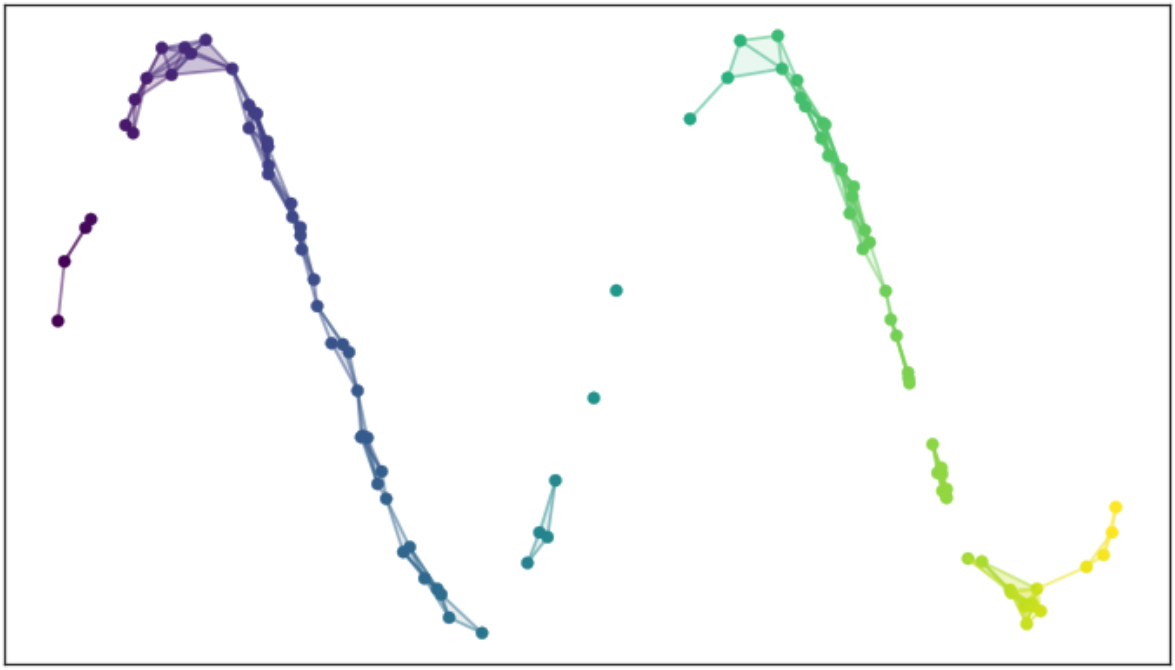
Por ejemplo, suponga que se tiene un conjunto con esta forma:



Si se toma cada punto y se dibuja una bola alrededor de él, se obtiene algo como esto:



Podemos generar un complejo simplicial a través de un complejo Vietoris-Rips. Este complejo se construye tomando cada bola y creando una arista entre cada par de bolas que se superponen. Luego, se crean triángulos entre cada terna de bolas que se superponen, y así sucesivamente.



Esto genera que ahora los datos estén representados a través de un grafo en baja dimensión.

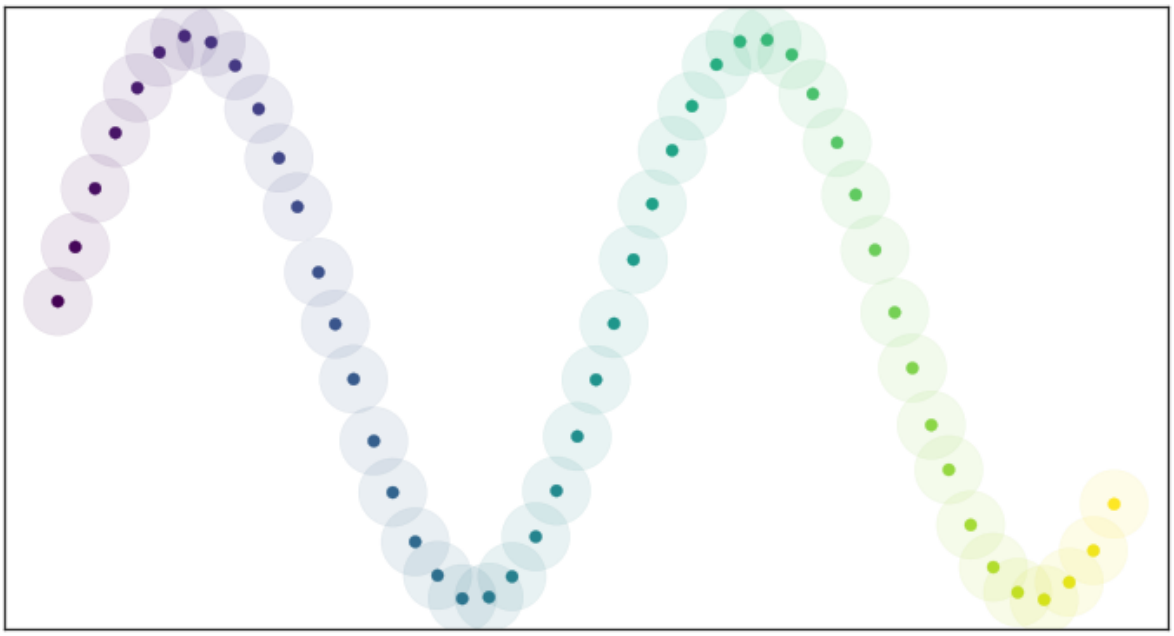
15.3.5 Adaptación del problema a datos reales

Problema #1: Escogencia del radio

La técnica anterior tiene un problema, no sabemos de antemano el radio óptimo de las bolas. Entonces:

- Radio es muy pequeno -> No se capturan las relaciones entre los puntos
- Radio es muy grande -> Se pierde la estructura local de los datos

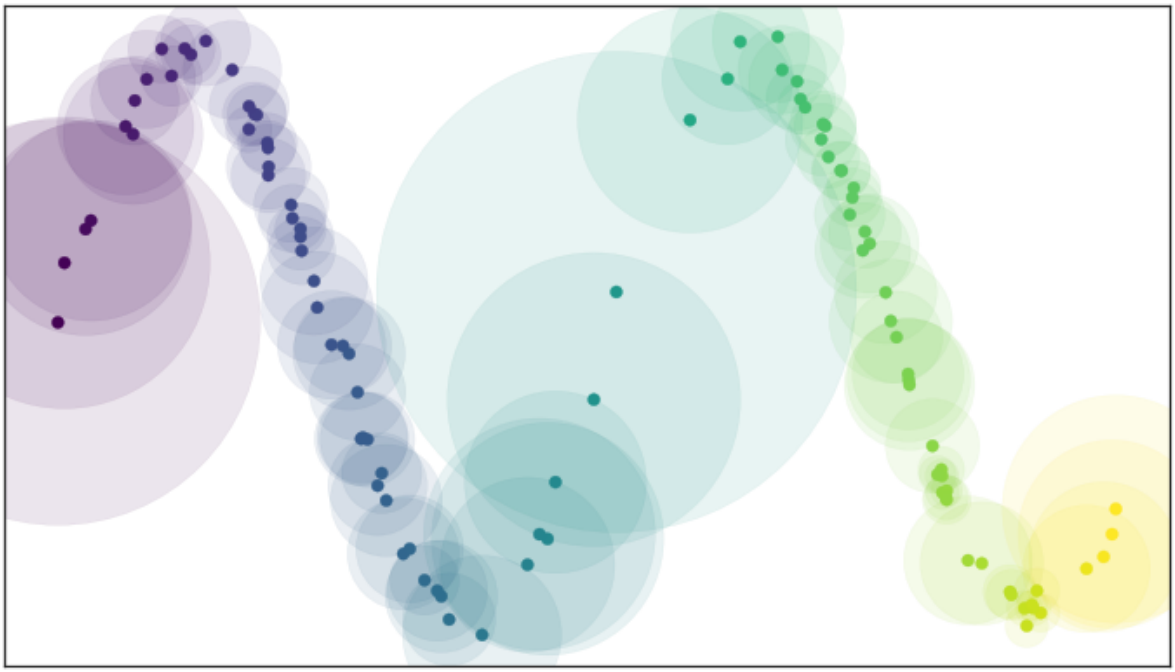
Solución: Asumir que los datos son uniformes en la variedad



El problema es que este tipo de supuesto no es real para toda la variedad. El problema es que la noción de distancia varía de punto a punto. En algunos puntos es más largo otros más corto.

Sin embargo, podemos construir una aproximación de uniformidad local de los puntos usando la geometría Riemaniana. Esto es que la bola alrededor de un punto se extiende hasta los k vecinos más cercanos. Así que cada punto tendrá su propia función de distancia.

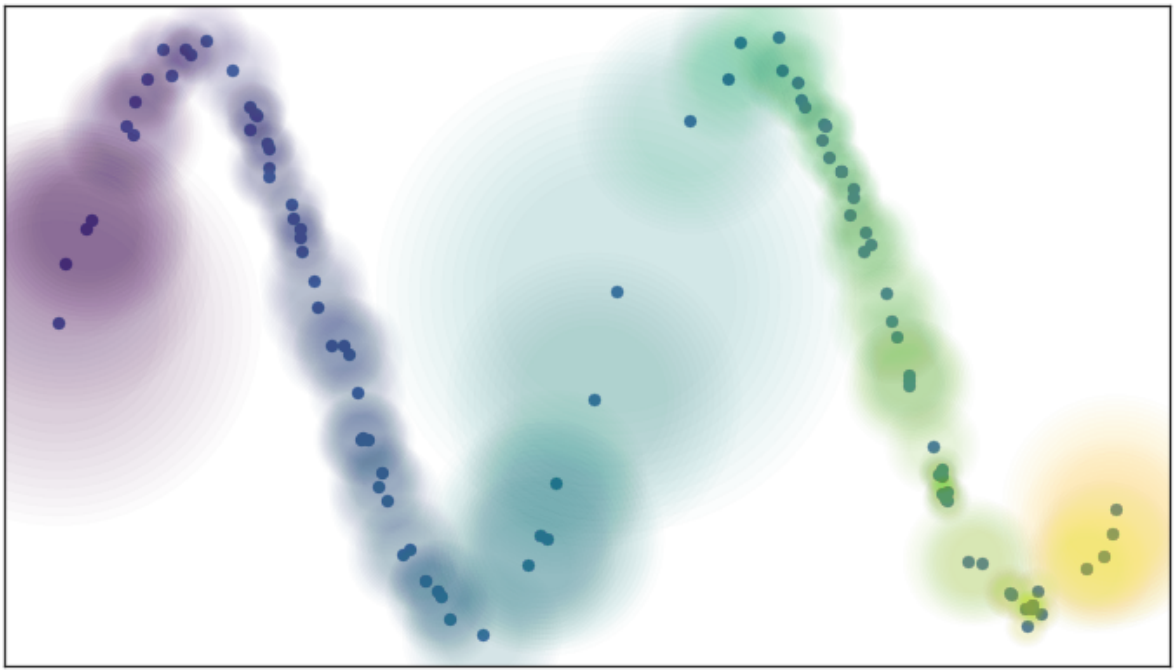
Desde un punto topológico, k significa qué tanto queremos estimar la métrica Riemaniana localmente. Si k es pequeño se explicaría features muy locales. Si k es grande, el features sería más global.



15.3.6 Un beneficio de la geometría Riemaniana

Se puede tener un espacio métrico asociado con cada punto. Es decir, cada punto puede medir distancia de forma significativa de modo que se puede estimar el peso de las aristas del grafo con las distancias que se genera.

Ahora, pensemos que si en lugar de decir que la covertura fue una un “si” o “no”, fuera un concepto más difuso como un valor de 0 a 1. Entonces, a partir del cierto punto, el valor se vuelve mas cercano a 0 conforme nos alejamos de este.



Problema #2: El manifold podría no estar conectado totalmente.

Es decir, el manifold podría ser simplemente un montón de islas de puntos sin vecinos muy cercanos.

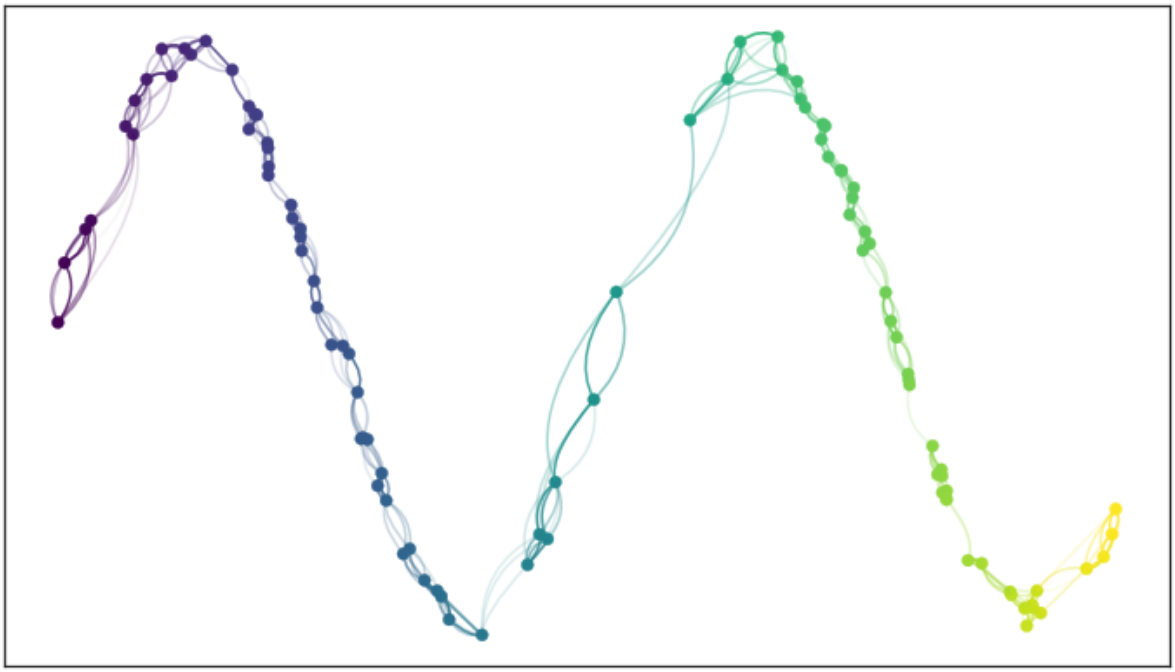
Solución: Usar la conectividad local.

El algoritmo asume que el manifold es **localmente conexo**. Debido a la maldición de la dimensionalidad, los datos en un espacio de alta dimensión tienen una mayor distancia, pero también pueden ser más similares entre sí. Esto significa que la distancia al primer vecino más cercano puede ser bastante grande, pero la distancia al décimo vecino más cercano suele ser solo ligeramente mayor (relativamente hablando). La restricción de conectividad local asegura que nos centremos en la diferencia de distancia entre los vecinos más cercanos, no en la distancia absoluta (lo que muestra que la diferencia entre vecinos es pequeña).

Problema 3: Incompatibilidad de la métrica local.

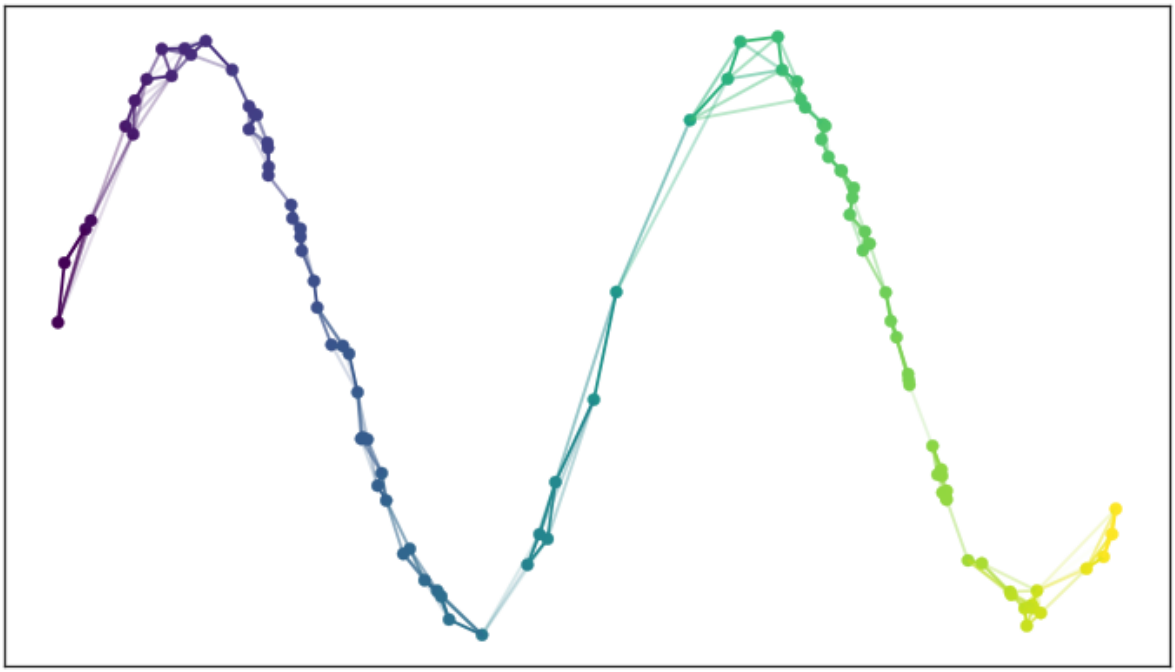
Cada punto tiene una métrica local asociada, y desde el punto de vista del punto a, la distancia desde el punto a hasta el punto b puede ser 1.5, pero desde el punto de vista del punto b, la distancia desde el punto b hasta el punto a podría ser solo 0.6.

Basándonos en la intuición del gráfico, se puede considerar que esto es un borde dirigido con diferentes pesos, como se muestra en la siguiente figura:



Combinar los dos bordes inconsistentes con pesos a y b juntos, entonces deberíamos tener un peso combinado $a + b - a \cdot b$. La forma de pensar esto es que el peso es en realidad la probabilidad de que exista el borde (1-símplex). Entonces, el peso combinado es la probabilidad de que exista al menos un borde.

Si aplicamos este proceso para fusionar todos los conjuntos simpliciales difusos juntos, terminamos con un solo complejo simplicial difuso, que podemos considerar nuevamente como un gráfico ponderado. En términos de cálculo, simplemente aplicamos la fórmula de combinación de pesos de bordes a todo el gráfico (el peso de los no bordes es 0). Al final, obtenemos algo como esto.



Entonces, asumiendo que ahora tenemos una representación topológica difusa de los datos (hablando matemáticamente, capturará la topología del manifold detrás de los datos), ¿cómo lo convertimos en una representación de baja dimensión?

15.3.7 Encontrando una representación de baja dimensión

La representación de baja dimensión debe tener la misma estructura topologica fuzzy de los datos. Tenemos dos problemas acá: 1. Cómo determinar la representación fuzzy en el espacio de baja dimensión y 2. cómo encontrar una buena.

Para 1., básicamente se haraá el mismo proceso pero con un espacio de $\mathbb{R}^2$ o $\mathbb{R}^3$.

Con 2., el problema se resuelve calibrando las mismas distancias de la topología difusa en la variedad con respecto a la distancias de la topología en $\mathbb{R}^2$.

Recordando el método de procesamiento de peso anterior, interpretamos el peso como la probabilidad de la existencia de un símplex. Dado que las dos topologías que estamos comparando comparten el mismo 0-símplex, es concebible que estamos comparando dos vectores de probabilidad indexados por el 1-símplex. Suponiendo que estos son todas variables de Bernoulli (el símplex final existe o no, y la probabilidad es un parámetro de la distribución de Bernoulli), la elección correcta aquí es la entropía cruzada.

Para entender el proceso priero definamos algunos conceptos.

Usando los k vecinos más cercanos para x_i es el conjunto de puntos $\{x_{i_1}, \ldots, x_{i_k}\}$ tal que:

$$\rho_i = \min_{1 \leq j \leq k} d(x_i, x_{i_j})$$

La función de peso para el 1-símplex $\{x_i, x_j\}$ es:

$$w_h(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j) - \rho_i}{\sigma_i}\right)$$

Si el conjunto de todos los posibles 1-símplexes entre x_i y x_j y la función ponderada hace que $w_h(x_i, x_j)$ sea el peso de ese simplex en la dimensión alta, y $w_l(x_i^l, x_j^l)$ en la dimensión baja, entonces la entropía cruzada es:

$$\sum_{i=1}^{N}\sum_{j=1}^{N} w_h(x_i, x_j)\frac{\log(w_h(x_i, x_j))}{\log(w_l(x_i^l, x_j^l))} + (1 - w_h(x_i, x_j))\frac{\log(1 - w_h(x_i, x_j))}{\log(1 - w_l(x_i^l, x_j^l))}$$

Desde la perspectiva de los gráficos, minimizar la entropía cruzada se puede considerar como un algoritmo de diseño de gráficos dirigido por fuerza.

El primer ítem, $w_h(e)\log(w_h(e)/w_l(e))$ proporciona atracción entre los puntos e cuando hay un peso mayor en el espacio de alta dimensión. Al minimizar este sumando $w_l(e)$ debe ser lo más grande posible y la distancia entre puntos es lo más pequeña posible.

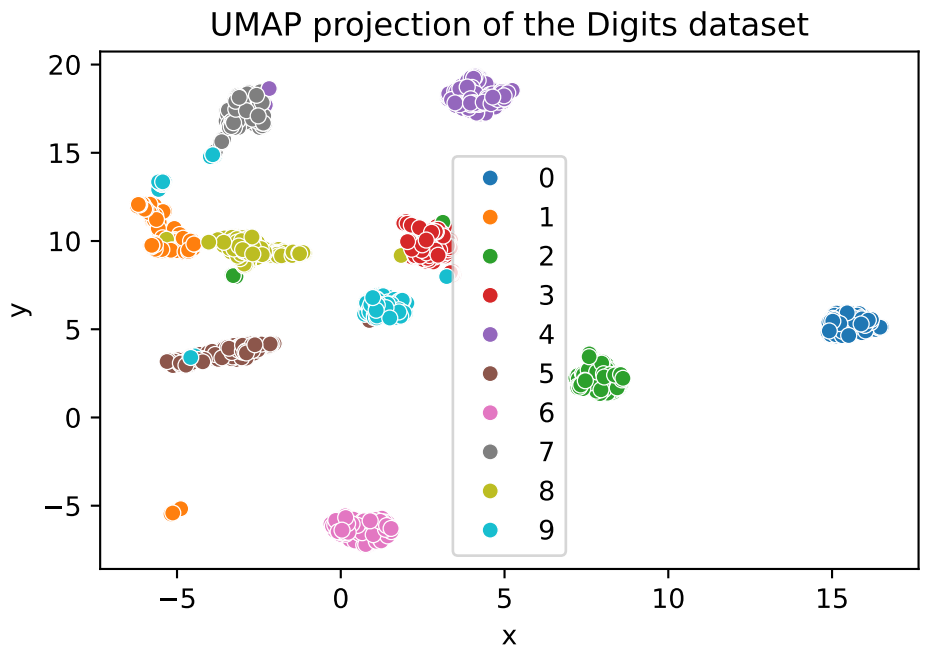
El segundo sumando, $(1 - w_h(e))\log((1 - w_h(e))/(1 - w_l(e)))$ proporciona fuerza repulsiva entre los dos segmentos de e cuando $w_h(e)$ es pequeño. Al hacer $w_l(e)$ lo más pequeño posible, se minimiza esta parte.

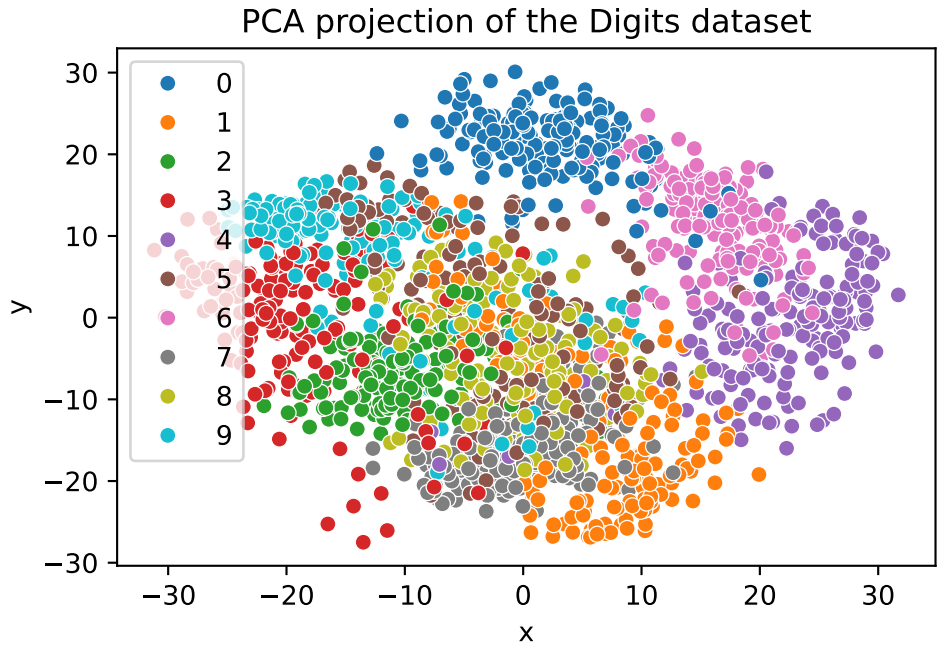
15.3.8 Ejemplos

```
import umap
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_digits
import pandas

# Cargar el conjunto de datos
digits = load_digits()
data = digits.data
target = digits.target

# Instanciar UMAP y reducir la dimensionalidad
reducer = umap.UMAP(random_state=42)
```

```
data_reduced = reducer.fit_transform(data)
data_reduced = pandas.DataFrame(data_reduced, columns=["x", "y"])
# Visualizar el resultado


sns.scatterplot(data= data_reduced,x = "x", y="y", hue=target, palette='tab10')
plt.title('UMAP projection of the Digits dataset')
plt.show()

# Compare el ressultado con PCA

from sklearn.decomposition import PCA

pca = PCA(n_components=2)
data_pca = pca.fit_transform(data)
data_pca = pandas.DataFrame(data_pca, columns=["x", "y"])

sns.scatterplot(data= data_pca,x = "x", y="y", hue=target, palette='tab10')
plt.title('PCA projection of the Digits dataset')
plt.show()
```

```
2024-02-12 14:46:59.688339: I tensorflow/core/platform/cpu_feature_guard.cc:182] This Tensorl
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebu
/usr/local/lib/python3.11/site-packages/umap/umap_.py:1943: UserWarning:

n_jobs value -1 overridden to 1 by setting random_state. Use no seed for parallelism.

OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels inste
```






```
import umap.umap_ as umap
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC

# Cargar el conjunto de datos
digits = load_digits()
data = digits.data
target = digits.target

# Dividir el conjunto de datos
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.25, random_sta

# Reducción de dimensionalidad con UMAP
umap_reducer = umap.UMAP(random_state=42)
X_train_reduced = umap_reducer.fit_transform(X_train)
X_test_reduced = umap_reducer.transform(X_test)


# Claificación con SVM
svm = SVC()
svm.fit(X_train_reduced, y_train)

# Predicción y evaluación
y_pred = svm.predict(X_test_reduced)
print("Accuracy con UMAP:", accuracy_score(y_test, y_pred))

# Clasificación con SVM
svm = SVC()
svm.fit(X_train, y_train)

# Predicción y evaluación
y_pred = svm.predict(X_test)
print("Accuracy sin UMAP:", accuracy_score(y_test, y_pred))
```

```
/usr/local/lib/python3.11/site-packages/umap/umap_.py:1943: UserWarning:

n_jobs value -1 overridden to 1 by setting random_state. Use no seed for parallelism.

Accuracy con UMAP: 0.9666666666666667
Accuracy sin UMAP: 0.9866666666666667
```