SCIKIT-LEARN

¿Qué es Scikit-Learn?

Scikit-Learn es una biblioteca de Python especializada en aprendizaje automático clásico (machine learning). Ofrece algoritmos ya implementados para clasificación, regresión, clustering y reducción de dimensionalidad, con un enfoque práctico y de uso rápido.

Las principales características de Scikit-Learn:

- Open source y gratuita.
- Fácil de aprender: código sencillo y estandarizado.
- Se integra con NumPy, Pandas y Matplotlib.
- Incluye datasets de prueba (Iris, Wine, Digits, etc.).
- Optimizada para modelos clásicos, no para redes neuronales profundas.

Módulos de aprendizaje automático

Los módulos de aprendizaje automático se utilizan para distintas tareas de aprendizaje automático, tales como clasificación, regresión o reducción de dimensionalidad.

Estos son algunos de los módulos más comunes de Scikit-Learn:

- Preprocesamiento de datos.
- Modelos de clasificación: Regresión Logística, Árboles de decisión, Nave Bayes y K-vecinos.
- Modelos de regresión: Regresión Lineal, Regresión Ridge, Regresión Lasso y Regresión polinómica.
- Agrupamiento: K-Means, DBSCAN y Mean Shift.
- Reducción de dimensionalidad: Análisis de Componentes Principales y Análisis de Discriminante Lineal.
- Validación del modelo.

¿Qué es la regresión lineal?

La regresión lineal es un algoritmo de aprendizaje supervisado utilizado para modelar la relación entre una variable dependiente (Y) y una o más variables independientes (X). El objetivo del modelo es encontrar la línea recta que mejor se ajusta a los datos y utilizarla para hacer predicciones sobre nuevos datos.

Tipo de problema	Algoritmo / Clase principal
Clasificación	LogisticRegression, KNeighborsClassifier, DecisionTreeClassifier, RandomForestClassifier, GradientBoostingClassifier, SVC, LinearSVC, GaussianNB, MultinomialNB, BernoulliNB, Perceptron, MLPClassifier
Regresión	LinearRegression, Ridge, Lasso, ElasticNet, DecisionTreeRegressor, RandomForestRegressor, GradientBoostingRegressor, SVR, KNeighborsRegressor, MLPRegressor
Clustering	KMeans, DBSCAN, MeanShift, AgglomerativeClustering, SpectralClustering, Birch
Reducción de dimensionalidad	PCA, KernelPCA, TruncatedSVD, FactorAnalysis, FastICA
Modelos de Ensamble	BaggingClassifier, BaggingRegressor, AdaBoostClassifier, AdaBoostRegressor, VotingClassifier, VotingRegressor, StackingClassifier, StackingRegressor
Preprocesamiento	StandardScaler, MinMaxScaler, OneHotEncoder, LabelEncoder
Selección de características	SelectKBest, RFE (Recursive Feature Elimination)
Validación de modelos	train_test_split, cross_val_score, GridSearchCV, RandomizedSearchCV

Datasets incluidos en Scikit-Learn

Dataset	Función para cargar	Tipo de problema	Descripción
Iris	load_iris()	Clasificación	Flores de iris, 3 clases, 150 muestras.
Digits	load_digits()	Clasificación	Imágenes 8x8 de dígitos escritos a mano (0–9).
Wine	load_wine()	Clasificación	Vinos italianos, 13 características químicas.
Breast Cancer	load_breast_cancer()	Clasificación binaria	Tumores malignos o benignos en mama.
Diabetes	load_diabetes()	Regresión	Datos médicos para predecir progresión de la diabetes.
Linnerud	load_linnerud()	Regresión multisalida	Ejercicios físicos (3 features) vs medidas corporales (3 salidas).

pip install scikit-learn

EJERCICIOS

Ejercicio 1

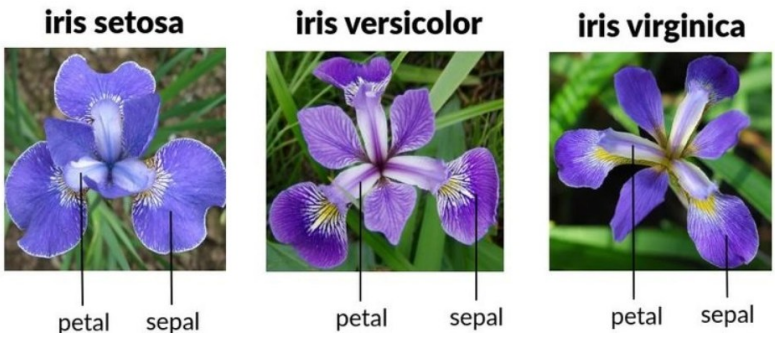
Implementa el siguiente algoritmo.

```
1   #Regresión Lineal con Scikit-learn
2
3   from sklearn.linear_model import LinearRegression
4
5   x = [[1], [2], [3], [4], [5]]
6   y = [5, 8, 12, 15, 16]
7
8   modelo=LinearRegression()
9   modelo.fit(x, y)
10  print(modelo.predict([[6]]))  # Predicción para x=6
```

¿Qué es el dataset Iris?

Scikit-learn trae un dataset muy famoso: Iris dataset. Contiene datos de flores Iris de 3 especies distintas:

- **A.** Setosa (0)
- **B.** Versicolor (1)
- **C.** Virginica (2)



Cada flor tiene **4 características numéricas** (medidas en centímetros):

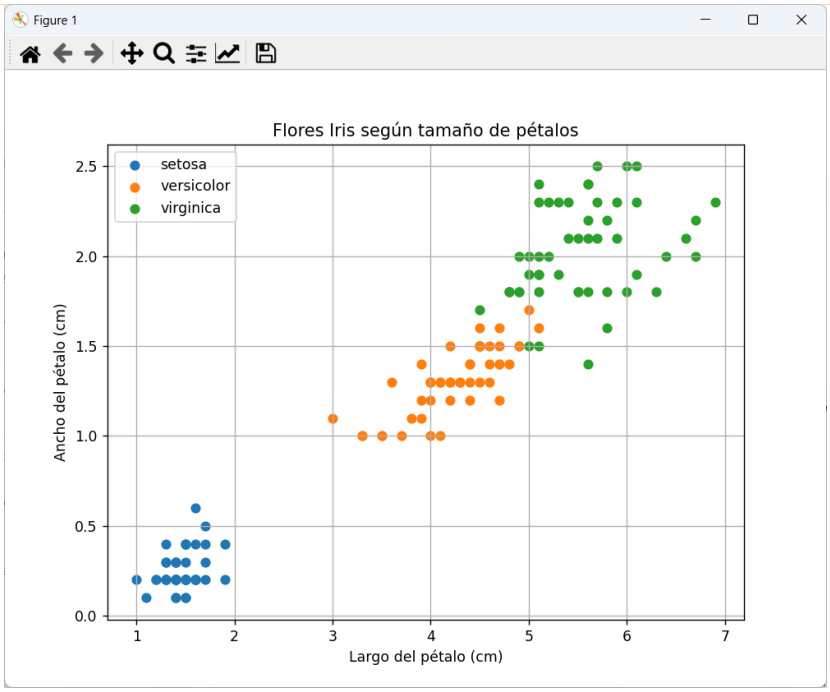
1. sepal length → Largo del sépalo
2. sepal width → Ancho del sépalo
3. petal length → Largo del pétalo
4. petal width → Ancho del pétalo

Ejercicio 2

Implementa el siguiente algoritmo.

C2_Scikit-learn_iris.py > ...

```python
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
import pandas as pd

#Cargar dataset
iris = load_iris()
X, y = iris.data, iris.target

#Convertir a DataFrame para verlo en tabla
df = pd.DataFrame(X, columns=iris.feature_names)
df["species"] = [iris.target_names[i] for i in y]

#Mostrar primeros datos
print("=== 10 primeros datos del dataset Iris ===")
print(df.head(10))

#Gráfico: comparar largo y ancho del pétalo
plt.figure(figsize=(8,6))
for species in iris.target_names:
    subset = df[df["species"] == species]
    plt.scatter(subset["petal length (cm)"], subset["petal width (cm)"], label=species)

plt.title("Flores Iris según tamaño de pétalos")
plt.xlabel("Largo del pétalo (cm)")
plt.ylabel("Ancho del pétalo (cm)")
plt.legend()
plt.grid(True)
plt.show()
```



Ejercicio 3

Implementa el siguiente algoritmo.

C3_monedas_scikit_learn_v1.py > ...

```python
import cv2
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

ruta = r"C:\Users\DELL\Desktop\python\senati\2_scikit-learn_pytorch\monedas.jpg"

# 1) Cargar imagen y convertir a gris
img = cv2.imread(ruta)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (15, 15), 0)

# 2) Detectar círculos
circles = cv2.HoughCircles(
    blur,
    cv2.HOUGH_GRADIENT,
    dp=1.2,
    minDist=50,
    param1=100,
    param2=30,
    minRadius=20,
    maxRadius=80
)

if circles is not None:
    circles = np.round(circles[0, :]).astype("int")
    radii = circles[:, 2].reshape(-1, 1)

    # 3) Clasificar tamaños de monedas
    kmeans = KMeans(n_clusters=5, random_state=0)
    labels = kmeans.fit_predict(radii)

    # 4) Definir colores para cada cluster
    colors = [(255,0,0), (0,255,0), (0,0,255), (255,255,0), (255,0,255)]

    # 5) Dibujar círculos con color según cluster
    for (x, y, r), label in zip(circles, labels):
        color = colors[label % len(colors)]  # Asigna color del cluster
        cv2.circle(img, (x, y), r, color, 2)
        cv2.putText(img, f"Clase {label}", (x - 20, y), cv2.FONT_HERSHEY_SIMPLEX,
                    0.6, color, 2)

    # Mostrar resultado
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    plt.axis("off")
    plt.show()
else:
    print("No se detectaron monedas.")
```

PYTORCH

1. ¿Qué es PyTorch?

PyTorch es una **biblioteca de aprendizaje automático de código abierto basada en la biblioteca de Torch**, utilizado para aplicaciones como **visión artificial y procesamiento de lenguajes naturales**, principalmente desarrollado por el Laboratorio de Investigación de Inteligencia Artificial de Facebook (FAIR). Es un software libre y de código abierto liberado bajo la Licencia Modificada de BSD. A pesar de que la interfaz de Python está más pulida y es el foco principal del desarrollo, PyTorch también tiene una interfaz en C++.

En resumen, PyTorch es una librería de Python muy popular para Inteligencia Artificial y Deep Learning. Se utiliza para crear, entrenar y probar redes neuronales. Es muy flexible y fácil de aprender, ideal para educación y proyectos reales.

2. Conceptos Clave

- **Tensors:** Son como arreglos (matrices) de Numpy, pero con superpoderes:
  - Pueden ejecutarse en CPU o GPU.
  - Son la base para todos los cálculos en redes neuronales.
- **Autograd (Derivadas automáticas):** PyTorch calcula automáticamente las derivadas, necesarias para entrenar redes neuronales.
- **Modelos (nn.Module):** En PyTorch, un modelo (red neuronal) se define como una clase o con nn.Sequential.
- **Loss Function (Función de pérdida):** Mide el error entre la predicción de la red y el valor real.
- **Optimizer:** Ajusta los parámetros de la red para reducir el error.

3. Flujo de trabajo

Todo proyecto con PyTorch sigue estos pasos:

3. Flujo de trabajo en PyTorch

Todo proyecto con PyTorch sigue estos pasos:

1. **Preparar datos**
   - Entradas (X) y salidas (Y).
2. **Definir el modelo**
   - Arquitectura de la red.
3. **Definir función de pérdida y optimizador**
   - Para medir errores y mejorar la red.
4. **Entrenar el modelo**
   - Ciclo: predicción → pérdida → retropropagación → actualización.
5. **Probar el modelo**
   - Verificar si aprendió bien.

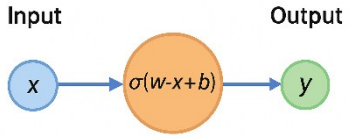
Ejercicio 1

Implementa el siguiente algoritmo.

P01_PyTorch_2x.py > ...

```python
import torch.nn as nn
import torch.optim as optim

# 1) Datos: y = 2x
X = torch.tensor([[1.0],[2.0],[3.0],[4.0]])
Y = torch.tensor([[2.0],[4.0],[6.0],[8.0]])

# 2) Modelo: una sola neurona (lineal)
model = nn.Linear(1, 1)

# 3) Función de pérdida y optimizador
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# 4) Entrenamiento
for epoch in range(200):
    Y_pred = model(X)          # Predicción
    loss = criterion(Y_pred, Y) # Error

    optimizer.zero_grad()       # Borrar gradientes
    loss.backward()             # Retropropagación
    optimizer.step()            # Ajustar pesos

    if (epoch+1) % 50 == 0:
        print(f"Época {epoch+1}, Pérdida: {loss.item():.4f}")

# 5) Probar la red
test = torch.tensor([[5.0]])
print("Para x=5, la red predice:", model(test).item())
```

1. Le damos ejemplos (1→2, 2→4, 3→6, 4→8).
2. La red trata de adivinar la regla.
3. Calculamos el error.
4. La red ajusta sus parámetros (pesos).
5. Al final, si probamos con x=5, debería dar una respuesta cerca de 10.



Ejercicio 2

Implementa el siguiente algoritmo.

P02_PyTorch_2x_dos_neuronas.py > ...

```python
import torch
import torch.nn as nn
import torch.optim as optim

# 1) Datos: y = 2x
X = torch.tensor([[1.0],[2.0],[3.0],[4.0]])
Y = torch.tensor([[2.0],[4.0],[6.0],[8.0]])

# 2) Modelo: ahora con una capa oculta de 2 neuronas
model = nn.Sequential(
    nn.Linear(1, 2),  # Capa oculta: 1 entrada -> 2 neuronas
    nn.ReLU(),        # Activación ReLU
    nn.Linear(2, 1)   # Capa de salida: 2 neuronas -> 1 salida
)

# 3) Función de pérdida y optimizador
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

# 4) Entrenamiento
for epoch in range(500):
    Y_pred = model(X)           # Predicción
    loss = criterion(Y_pred, Y) # Error

    optimizer.zero_grad()       # Borrar gradientes
    loss.backward()             # Retropropagación
    optimizer.step()            # Ajustar pesos

    if (epoch+1) % 100 == 0:
        print(f"Época {epoch+1}, Pérdida: {loss.item():.4f}")

# 5) Probar la red
test = torch.tensor([[5.0]])
print("Para x=5, la red predice:", model(test).item())
```

- La primera capa transforma x en **2 valores internos (neuronas ocultas)**.
- Se aplica la función **ReLU** (que deja pasar valores positivos y bloquea negativos).
- La segunda capa convierte esas 2 neuronas en la salida final.