02. Herramientas/Librerías de Python para trabajar con IA

Librería	Propósito
numpy	Cálculos numéricos
pandas	Manipulación y análisis de datos
matplotlib	Visualización de datos
scikit-learn	Implementar algoritmos de aprendizaje automático

Instalación de dependencias

Hasta ahora hemos visto que para instalar dependencias tenemos que hacerlo con **pip install nombre_modulo** (Ej. `pip install requests`)

Podemos crear un archivo en el raíz de nuestro proyecto llamado `requirements.txt` y a través de él instalar todos los módulos de una vez.

```
pip install -r requirements.txt
```

📒 Jupyter Notebook

Jupyter Notebook es una herramienta poderosa para trabajar con código interactivo y documentado. Su flexibilidad lo hace ideal para ciencia de datos, enseñanza y desarrollo de IA.

Es una aplicación web interactiva.

Permite crear y compartir documentos que contienen código en vivo, ecuaciones, visualizaciones y texto explicativo.

Ampliamente utilizado en análisis de datos, *Machine Learning*, computación científica y enseñanza de programación.

ESTRUCTURA

Básicamente un notebook de Jupyter consta únicamente de **celdas**. Éstas pueden contener:

Código: Ejecutan código en vivo en el lenguaje soportado (por defecto Python)

Markdown: Agregar texto con formato, ecuaciones **LaTex**,

El núcleo principal de Jupyter es el “**kernel**”, que ejecuta el código y devuelve los resultados.

Es compatible con varios lenguajes de programación, aunque el más común es Python.

imágenes y enlaces. Wikipedia.

Salida: Contiene los resultados de ejecución del código, como gráficos o texto.

Cada celda se ejecuta de forma independiente y muestra su salida justo debajo.

Jupyter Notebook en VSCode

Ctrl + may + p
>jupyter new notebook

1️⃣ Instala las extensiones de **Python** y **Jupiter**.

2️⃣ Crea un Python Notebook. Tendrás que configurar el entorno de ejecución (recomendable el entorno virtual)

3️⃣ La primera vez que ejecutes un código es posible que te invite a instalar **ipykernel**.

> ℹ️ Los archivos Jupiter Notebook tienen la extensión **.ipynb** (*Interactive Python Notebook*)

🧮 numpy

Explorando NumPy para computación científica en Python

Introducción a NumPy

NumPy (**Numerical Python**) es una biblioteca fundamental para la computación científica o cálculo numérico en Python.

Proporciona soporte para matrices y grandes arreglos multidimensionales.

Usa técnicas de indexación y segmentación.

Creando arreglos NumPy

Los arreglos pueden ser creados usando la función `numpy.array()` .

Habilita el almacenamiento y la manipulación de datos numéricos de manera eficiente.

Características principales de NumPy

Arreglos: El núcleo de NumPy está en **ndarray** (*N-dimensional array*), un contenedor rápido y flexible para manejar grandes conjuntos de datos eficientemente.

Vectorización: Permite operaciones elemento por elemento en arreglos sin la necesidad de bucles.

Funciones matemáticas: Ofrece un amplio rango de funciones matemáticas para tareas como trigonometría, estadísticas, álgebra lineal, etc.

Broadcasting: Facilita operaciones en matrices de diferentes tamaños, permitiendo operaciones aritméticas entre arreglos y escalares.

Operaciones básicas con NumPy

Soporta varias operaciones como operaciones aritméticas, cálculos estadísticos, manipulación de arreglos, etc.

Indexación y segmentación de matrices

Permite acceso de manera fácil y manipulación de elementos en arreglos multidimensionales.

NumPy es importantísima. Unas 400 librerías dependen de esta librería. Una de las más importantes que depende de NumPy es TensorFlow, la biblioteca por excelencia para trabajar con Redes Neuronales.

```
import numpy as np

##### 1. CREACIÓN DE ARRAYS #####

# Unidimensionales
a = np.array([1, 2, 3, 4, 5])
print(a)  # Output: [1 2 3 4 5]
print(type(a))  # Output: <class 'numpy.ndarray'>

# Multidimensionales
b = np.array([[1, 2, 3], [4, 5, 6]])
print(b)
# Output:
```

```
# [[1 2 3]
#  [4 5 6]]

##### 2. PROPIEDADES DE LOS ARRAY #####
print(a.shape)  # Dimensiones del array
print(a.ndim)   # Número de dimensiones
print(a.dtype)  # Tipo de datos de los elementos
print(a.size)   # Número total de elementos
print(a.itemsize)  # Tamaño en bytes de cada elemento

##### 3. INICIALIZACIÓN DE ARRAYS ESPECIALES #####
np.zeros((2, 3))   # Matriz de ceros de 2×3
np.ones((3, 4))    # Matriz de unos de 3×4
np.full((2, 2), 7) # Matriz llena con un valor específico
np.eye(3)          # Matriz identidad de 3×3
np.random.rand(2, 3)  # Matriz de números aleatorios

##### 4. INDEXACIÓN Y SLICING EN ARRAYS #####
arr = np.array([[10, 20, 30], [40, 50, 60]])

print(arr[0, 1])  # Acceder a un elemento específico (fila 0, columna 1)
print(arr[:, 1])  # Seleccionar toda la columna 1
print(arr[0, :])  # Seleccionar toda la fila 0
print(arr[1, 1:]) # Slicing: desde la columna 1 hasta el final de la fila 1

##### 5. OPERACIONES MATEMÁTICAS #####
# NumPy permite realizar operaciones matemáticas sobre arrays sin necesi
dad de iteraciones explícitas (vectorización).

# Elemento a elemento
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])

print(x + y)  # Suma
print(x - y)  # Resta
print(x * y)  # Multiplicación elemento a elemento
print(x / y)  # División
print(x ** 2) # Potencia
```

```
# Funciones Universales (ufuncs)
np.sqrt(x)  # Raíz cuadrada
np.exp(x)   # Exponencial
np.log(x)   # Logaritmo natural
np.sin(x)   # Seno
np.cos(x)   # Coseno
np.tan(x)   # Tangente

##### 6. ÁLGEBRA LINEAL #####
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(np.dot(A, B))    # Producto matricial
print(np.linalg.inv(A)) # Matriz inversa
print(np.linalg.det(A)) # Determinante de la matriz
print(np.linalg.eig(A)) # Valores y vectores propios

##### 7. MANEJO DE DATOS FALTANTES #####
# Filtrado basado en condiciones
arr = np.array([1, 2, 3, 4, 5, 6])
print(arr[arr > 3])  # Filtrar valores mayores a 3

# Manejo de datos faltantes
arr = np.array([1, 2, np.nan, 4])
print(np.isnan(arr))  # Devuelve una máscara booleana
print(arr[~np.isnan(arr)])  # Filtrar valores no NaN

##### 8. BROADCASTING (rmite realizar operaciones entre arrays de difer
entes formas sin necesidad de expandirlos explícitamente) #####
# Suma con un escalar
A = np.array([[1, 2], [3, 4]])
print(A + 10)

# Sumar un vector fila a una matriz
B = np.array([10, 20])
print(A + B)  # Broadcasting automático
```

🐼 pandas

Pandas para análisis y manipulación de datos

Series Pandas

Es un arrego unidimensional de datos con etiquetas capaz de manejar varios tipos de datos tales como enteros, cadenas y números de puntos flotantes.

Integración

Pandas se integra sin problemas con otras bibliotecas como NumPy, Matplotlib y otras, haciéndola una herramienta versátil para el análisis de datos en Python.

Alineación de datos

Pandas alinea automáticamente los datos para las operaciones, manejando los datos faltantes con elegancia para garantizar un análisis preciso.

DataFrames Pandas

Es una estructura de datos tabulares bidimensional y de tamaño mutable en Pandas con ejes etiquetados para filas y columnas.

Herramientas de Manipulación de datos

Pandas proporciona herramientas poderoas para filtado de datos, agrupación, combinación, reoroentación y otras para manipular y analizar datos eficientemente.

Manejo de datos en Python: cargando y limpiando los datos

Cargando datos

Utiliza la librería `pandas` para cargar datos en Python de varias fuentes, como archivos CSV, Excel, bases de datos SQL y la web.

Usa el objeto `DataFrame` de `pandas` para trabajar eficientemente con

Técnicas de limpieza de datos

Implementa técnicas de limpieza de datos tales como manejo de valores faltantes, eliminación de duplicados, y lidiar con datos atípicos (erróneos o incosistentes) para asegurar la calidad de los datos.

estructuras de datos tabulares para el análisis.

Utiliza las funciones y métodos de `pandas` para limpiar y preparar los conjuntos de datos para mayor análisis y procesamiento.

Limpieza de datos

Detecta y maneja valores faltantes en los datos utilizando métodos como `dropna()` para eliminar filas con valores faltantes o `fillna()` para reemplazarlos con valores específicos.

Identifica y elimina filas duplicadas para asegurarse la presición y consistencia de los datos.

Abordar los valores atípicos (outliers) detectando y manejando puntos de datos que difieren significativamente del resto del conjunto de datos.

Importancia de la limpieza de datos

La limpieza de datos es esencial para garantizar la calidad y confiabilidad de los datos utilizados antes de realizar las tareas de análisis y modelado.

Al dominar las técnicas de limpieza de datos, puede mejorar la calidad de sus datos y extraer información significativa de manera eficaz.

Técnicas de identificación de valores atípicos y manipulación de datos

Identificación de valores atípicos

Los valores atípicos son valores que se alejan significativamente de la distribución de los datos.

Un método común para identificar valores atípicos es el método de las puntuaciones Z (Z-score), donde valores absolutos en la puntuación Z por encima de 3 se consideran atípicos (o muy por encima de la media o muy por debajo).

Técnicas de manipulación de datos

La manipulación de datos implica la transformación y limpieza de los

Manejando valores atípicos

Una vez identificados los valores atípicos, deben manejarse de manera adecuada para prevenir distorsiones en los análisis o modelos.

La eliminación de valores con Z-scores mayores que 3 puede ayudar en la limpieza de los datos efectivamente.

datos en un formato más adecuado para el análisis.

Técnicas como filtrado, ordenamiento, agrupación y agregación a la vez que la aplicación de funciones a los datos pueden ayudar a extraer información valiosa y significativa del conjunto de datos.

```
import pandas as pd

##### 1. ESTRUCTURAS DE DATOS EN PANDAS #####

#--- Series ---
datos = [10, 20, 30, 40]
serie = pd.Series(datos, index=['a', 'b', 'c', 'd'])
print(serie)

# Salida
# a    10
# b    20
# c    30
# d    40
# dtype: int64

# Se puede acceder a los elementos como un diccionario
print(serie['b'])  # 20

#--- DataFrame ---
datos = {
    'Nombre': ['Ana', 'Luis', 'Pedro', 'Marta'],
    'Edad': [25, 30, 35, 40],
    'Ciudad': ['Madrid', 'Barcelona', 'Valencia', 'Sevilla']
}

df = pd.DataFrame(datos)
print(df)
```

```
# Salida
#    Nombre  Edad    Ciudad
# 0    Ana    25     Madrid
# 1   Luis    30  Barcelona
# 2  Pedro    35   Valencia
# 3  Marta    40    Sevilla

##### 2. OPERACIONES BÁSICAS EN PANDAS #####

# Carga de datos
df = pd.read_csv("archivo.csv")  # Desde un CSV
df = pd.read_excel("archivo.xlsx")  # Desde un Excel
df = pd.read_json("archivo.json")  # Desde un JSON

# Exploración de los datos
print(df.head())  # Primeras 5 filas
print(df.tail())  # Últimas 5 filas
print(df.info())  # Información general
print(df.describe())  # Estadísticas de las columnas numéricas

# Acceso a filas y columnas
print(df['Nombre']) # Selección de la columna "Nombre"
print(df.loc[2])  # Fila con índice 2
print(df.iloc[1:3])  # Filas desde el índice 1 hasta 2 (sin incluir el 3)

##### 3. MANIPULACIÓN DE DATOS #####

# Filtrado de datos (lo tengo que guardar en una variable porque el filtro no
se guarda en el set de datos
df_filtrado = df[df['Edad'] > 30]
print(df_filtrado)

# Ordenamiento
df_ordenado = df.sort_values(by='Edad', ascending=False)
print(df_ordenado)

# Modificación de datos
df['Edad'] = df['Edad'] + 1  # Incrementar en 1 todas las edades
```

```
df['Salario'] = [3000, 4000, 3500, 4500] # Agregando una nueva columna
df.drop(columns=['Ciudad'], inplace=True) # Eliminando una columna

##### 4. MANEJO DE DATOS NULOS #####

# Identificar valores nulos
print(df.isnull().sum())  # Cuenta de valores nulos por columna

# Eliminar filas o columnas con nulos
df.dropna(inplace=True)  # Elimina filas con valores nulos
df.dropna(axis=1, inplace=True)  # Elimina columnas con valores nulos

# Rellenar Valores Nulos
df.fillna(value=0, inplace=True)  # Rellena nulos con 0
df['Edad'].fillna(df['Edad'].mean(), inplace=True)  # Rellena con el promedi
o

##### 5. AGRUPACIONES Y OPERACIONES AVANZADAS #####

# Agrupamiento de datos
df.groupby('Ciudad')['Edad'].mean() # Saca la media de edad por ciudades

# Aplicación de funciones
df['Edad'] = df['Edad'].apply(lambda x: x * 2)  # Multiplica todas las edades
por 2

# Combinación de DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Nombre': ['Ana', 'Luis', 'Pedro']})
df2 = pd.DataFrame({'ID': [1, 2, 3], 'Salario': [3000, 4000, 5000]})

df_merge = pd.merge(df1, df2, on='ID')
print(df_merge)

##### 6. EXPORTAR DATOS #####
df.to_csv("salida.csv", index=False)
df.to_excel("salida.xlsx", index=False)
df.to_json("salida.json")
```

```
##### 7. VISUALIZACIÓN CON PANDAS #####
import matplotlib.pyplot as plt

df['Edad'].plot(kind='bar')
plt.show()
```

(02/07/2025)

📊 matplotlib 🎨

Matplotlib es una de las bibliotecas más utilizadas en Python para crear visualizaciones de datos. Permite generar gráficos en 2D y se integra fácilmente con otras bibliotecas como NumPy y Pandas.

1️⃣ Instalación

Si aún no tienes Matplotlib instalado, puedes hacerlo con:

```
pip install matplotlib
```

2️⃣ Importación y Estructura Básica

Para comenzar a usar Matplotlib, generalmente se importa de la siguiente manera:

```
import matplotlib.pyplot as plt
```

El módulo `pyplot` proporciona una interfaz similar a MATLAB, con funciones para crear y modificar gráficos fácilmente.

3️⃣ Creación de un Gráfico Básico

Aquí tienes un ejemplo de cómo crear un gráfico de líneas simple:

```
import matplotlib.pyplot as plt

# Datos
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
```

```
# Crear gráfico
plt.plot(x, y, marker='o', linestyle='-', color='b', label="Ventas")

# Personalizar
plt.xlabel("Días")
plt.ylabel("Ventas")
plt.title("Ventas Diarias")
plt.legend()  # Mostrar leyenda
plt.grid(True)  # Agregar cuadrícula

# Mostrar gráfico
plt.show()
```

🔷 **Explicación:**

- `plt.plot(x, y, ...)` → Grafica los datos `x` contra `y`.
- `marker='o'` → Pone marcadores en los puntos.
- `linestyle='-'` → Define el estilo de la línea.
- `color='b'` → Usa color azul (`b`).
- `label="Ventas"` → Etiqueta para la leyenda.
- `plt.xlabel()` / `plt.ylabel()` → Etiquetas para los ejes.
- `plt.title()` → Título del gráfico.
- `plt.legend()` → Muestra la leyenda.
- `plt.grid(True)` → Activa la cuadrícula.

4️⃣ Tipos de Gráficos en Matplotlib

📈 1. Gráfico de Líneas

```
plt.plot(x, y, marker='o', linestyle='-', color='r')
plt.show()
```

✔️ Útil para mostrar tendencias.

📊 2. Gráfico de Barras

```
categorias = ["A", "B", "C", "D"]
valores = [10, 20, 15, 25]

plt.bar(categorias, valores, color='purple')
plt.xlabel("Categorías")
plt.ylabel("Valores")
plt.title("Gráfico de Barras")
plt.show()
```

✔️ Útil para comparar datos.

🎂 3. Gráfico de Pastel

```
etiquetas = ["Manzanas", "Bananas", "Cerezas"]
valores = [30, 50, 20]

plt.pie(valores, labels=etiquetas, autopct='%1.1f%%', colors=['red', 'yello
w', 'pink'])
plt.title("Distribución de Frutas")
plt.show()
```

✔️ Muestra proporciones.

📊 4. Histograma

```
import numpy as np

datos = np.random.randn(1000)  # Genera 1000 valores aleatorios
plt.hist(datos, bins=30, color='green', edgecolor='black')
plt.xlabel("Valores")
plt.ylabel("Frecuencia")
plt.title("Histograma")
plt.show()
```

✔️ Útil para distribuciones de datos.

🌐 5. Gráfico de Dispersión

```
import numpy as np

x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y, color='blue')
plt.xlabel("Eje X")
plt.ylabel("Eje Y")
plt.title("Gráfico de Dispersión")
plt.show()
```

✔️ Muestra relaciones entre variables.

5️⃣ Personalización Avanzada

Matplotlib permite personalizar gráficos en profundidad.

📌 Cambiar Tamaño de la Figura

```
plt.figure(figsize=(8, 6))  # Ancho 8, Alto 6
plt.plot(x, y)
plt.show()
```

📌 Agregar Líneas de Referencia

```
plt.axhline(y=20, color='r', linestyle='--')  # Línea horizontal
plt.axvline(x=3, color='g', linestyle='-.')  # Línea vertical
plt.show()
```

📌 Subgráficos (Múltiples Gráficos en una Figura)

```
fig, ax = plt.subplots(1, 2, figsize=(10, 4))

ax[0].plot(x, y, color='b')
ax[0].set_title("Gráfico 1")
```

```
ax[1].bar(categorias, valores, color='orange')
ax[1].set_title("Gráfico 2")

plt.show()
```

✔️ Para comparar visualizaciones.

6️⃣ Guardar Gráficos

Puedes guardar gráficos en archivos de imagen con:

```
plt.savefig("grafico.png", dpi=300)  # DPI controla la calidad
```

Puedes guardar en formatos `.png`, `.jpg`, `.pdf`, etc.

7️⃣ Matplotlib y Pandas

Si usas `pandas`, puedes integrar Matplotlib fácilmente:

```
import pandas as pd

# Crear DataFrame
df = pd.DataFrame({
    "Día": [1, 2, 3, 4, 5],
    "Ventas": [10, 20, 25, 30, 40]
})

# Graficar
df.plot(x="Día", y="Ventas", kind="line", marker="o", color="r")
plt.title("Ventas por Día")
plt.show()
```

✔️ Se integra perfectamente con Pandas.

8️⃣ Alternativas a Matplotlib

Si buscas otras opciones más avanzadas:

- **Seaborn** → Para gráficos más estéticos.

- **Plotly** → Gráficos interactivos.
- **Bokeh** → Visualizaciones web interactivas.

🎯 Conclusión

Matplotlib es una herramienta poderosa para la visualización de datos en Python. Aunque su sintaxis puede parecer extensa, permite crear gráficos muy detallados y personalizables.

👨‍🔬 scikit-learn

Scikit-learn es una de las bibliotecas más populares en Python para el aprendizaje automático (Machine Learning). Se basa en otras bibliotecas fundamentales de Python como **NumPy, SciPy y Matplotlib**, y proporciona herramientas eficientes y fáciles de usar para tareas de clasificación, regresión, clustering, reducción de dimensionalidad, selección de modelos y preprocesamiento de datos.

---

1. Características Principales

Scikit-learn se distingue por varias características clave:

- **Simplicidad y facilidad de uso**: La API de Scikit-learn es consistente y fácil de aprender, lo que permite entrenar modelos en pocas líneas de código.
- **Basado en NumPy, SciPy y Matplotlib**: Aprovecha la eficiencia de estas bibliotecas para realizar cálculos numéricos y visualizaciones.
- **Gran variedad de algoritmos**: Soporta algoritmos de clasificación, regresión, clustering y reducción de dimensionalidad.
- **Compatibilidad con datos en forma de matrices**: Utiliza estructuras de datos similares a los arrays de NumPy para facilitar el procesamiento.
- **Buen manejo de la selección y evaluación de modelos**: Ofrece herramientas para dividir conjuntos de datos, validación cruzada y métricas de desempeño.
- **Soporte para preprocesamiento de datos**: Permite normalizar, escalar y transformar datos de manera sencilla.

- **Implementación eficiente**: Usa Cython y otras optimizaciones para mejorar el rendimiento.

---

2. Instalación de Scikit-Learn

Para instalar Scikit-learn, puedes usar `pip`:

```
pip install scikit-learn
```

También se recomienda instalarlo dentro de un entorno virtual para evitar conflictos con otras bibliotecas.

Si usas `conda`, puedes instalarlo con:

```
conda install scikit-learn
```

---

3. Componentes Clave de Scikit-Learn

Scikit-learn se basa en cinco pilares fundamentales:

3.1. Conjuntos de Datos

Scikit-learn incluye varios conjuntos de datos para pruebas y aprendizaje, como:

- `datasets.load_iris()`: Datos de flores Iris.
- `datasets.load_digits()`: Imágenes de dígitos escritos a mano.
- `datasets.load_wine()`: Datos sobre vinos.

Ejemplo de carga de datos:

```
from sklearn import datasets

iris = datasets.load_iris()
X, y = iris.data, iris.target
print(X.shape, y.shape)
```

También permite cargar datos desde archivos CSV o bases de datos con `pandas`.

3.2. Preprocesamiento de Datos

El preprocesamiento es esencial en machine learning. Scikit-learn ofrece herramientas para:

- **Escalado de datos:** `StandardScaler()`, `MinMaxScaler()`
- **Transformación de datos categóricos:** `OneHotEncoder()`, `LabelEncoder()`
- **Manejo de datos faltantes:** `SimpleImputer()`

Ejemplo de normalización de datos:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

3.3. Modelos de Aprendizaje Automático

Scikit-learn ofrece modelos para:

Clasificación

- `LogisticRegression()`: Regresión logística.
- `SVC()`: Máquinas de soporte vectorial (SVM).
- `KNeighborsClassifier()`: K-Nearest Neighbors (KNN).
- `RandomForestClassifier()`: Bosques aleatorios.

Ejemplo de clasificación con SVM:

```
from sklearn.svm import SVC

model = SVC(kernel='linear')
model.fit(X, y)
predictions = model.predict(X)
```

Regresión

- `LinearRegression()`: Regresión lineal.
- `DecisionTreeRegressor()`: Árboles de decisión.

- `SVR()`: Regresión con soporte vectorial.

Ejemplo de regresión lineal:

```
from sklearn.linear_model import LinearRegression

regressor = LinearRegression()
regressor.fit(X, y)
y_pred = regressor.predict(X)
```

Clustering

- `KMeans()`: Algoritmo k-means.
- `DBSCAN()`: Agrupamiento basado en densidad.
- `AgglomerativeClustering()`: Clustering jerárquico.

Ejemplo de clustering con K-Means:

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
labels = kmeans.labels_
```

Reducción de Dimensionalidad

- `PCA()`: Análisis de Componentes Principales.
- `LDA()`: Análisis Discriminante Lineal.
- `t-SNE()`: Embedding para visualización.

Ejemplo con PCA:

```
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

3.4. Selección y Evaluación de Modelos

Scikit-learn proporciona herramientas para evaluar y optimizar modelos:

División de datos en entrenamiento y prueba

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Validación cruzada

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(model, X, y, cv=5)
print(scores.mean())
```

Métricas de Evaluación

- `accuracy_score()`: Precisión en clasificación.
- `mean_squared_error()`: Error cuadrático medio en regresión.
- `classification_report()`: Reporte detallado de clasificación.

Ejemplo de evaluación de clasificación:

```
from sklearn.metrics import accuracy_score, classification_report

y_pred = model.predict(X_test)
print(accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

3.5. Pipeline y Automatización

Los **pipelines** permiten encadenar transformaciones y modelos de forma eficiente:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
```

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])

pipeline.fit(X_train, y_train)
```

4. Comparación con Otras Bibliotecas

Característica	Scikit-learn	TensorFlow	PyTorch
Modelos ML Clásicos	✅ Sí	❌ No	❌ No
Redes Neuronales	❌ No	✅ Sí	✅ Sí
Facilidad de uso	✅ Muy fácil	🔶 Medio	🔶 Medio
Computación en GPU	❌ No	✅ Sí	✅ Sí

Scikit-learn es ideal para modelos de machine learning clásicos, mientras que TensorFlow y PyTorch están diseñados para deep learning.

5. Conclusión

Scikit-learn es una herramienta poderosa y versátil para aprendizaje automático clásico. Sus ventajas incluyen:

✅ Fácil de usar

✅ Gran variedad de modelos

✅ Buenas herramientas de evaluación

✅ Ideal para proyectos de ML sin redes neuronales

Si quieres trabajar con redes neuronales, considera usar TensorFlow o PyTorch. Para tareas más generales de ML, Scikit-learn es una opción excelente.