

De Excel a Python: Análisis Inteligente de Datos con Machine Learning

Instructivo para manejo de librerías y flujo de trabajo para la elaboración y ejecución de proyectos con técnicas de machine learning

ÍNDICE

1. Librería Pandas

- 1.1 Principales comandos de pandas
- 1.2 ¿Qué es `df`?
- 1.3 Ejemplo simple con pandas

2. Librería NumPy

- 2.1 ¿Qué es NumPy?
- 2.2 Ejemplo simple con NumPy

3. Librería scikit-learn (sklearn)

- 3.1 Principales comandos de scikit-learn
- 3.2 ¿Qué es StandardScaler?
- 3.3 ¿Qué es LabelEncoder?
- 3.4 ¿Qué es scikit-learn?
- 3.5 `np.random.seed(42)` - Explicación
- 3.6 Ejemplo simple con scikit-learn (Regresión)
- 3.7 Ejemplo simple con scikit-learn - Clasificación con Random Forest

4. Librería openpyxl

- 4.1 Principales comandos de openpyxl
- 4.2 ¿Qué es openpyxl?
- 4.3 Ejemplo simple con openpyxl

5. Modelos PKL (Pickle) y Joblib

- 5.1 ¿Qué son los modelos PKL (Pickle) y por qué se guardan?
- 5.2 ¿Qué es Pickle?
- 5.3 ¿Qué contiene un modelo PKL?
- 5.4 Alternativas a Pickle
- 5.5 ¿Qué es un Framework?
- 5.6 Ejemplo práctico: Guardar y cargar modelos PKL
- 5.7 Flujo completo de guardado y carga
- 5.8 Ejemplo de despliegue en producción
- 5.9 Casos de uso comunes para modelos PKL
- 5.10 Mejores prácticas

- 5.11 Resumen final: Pickle vs Joblib

6. Librería PyTorch (Opcional)

- 6.1 Principales comandos de PyTorch
- 6.2 ¿Qué es PyTorch?
- 6.3 Ejemplo simple con PyTorch
- 6.4 Comparación: PyTorch vs scikit-learn

Este instructivo tiene como objetivo servir como guía de referencia para el curso **Excel a Python: Análisis Inteligente de Datos con Machine Learning**.

A lo largo del curso se realizará una revisión práctica de las librerías **Pandas**, **NumPy**, **Scikit-learn** y **openpyxl**, enfocada en la construcción, entrenamiento y despliegue de modelos de Machine Learning directamente integrados con Excel.

También se explicará la estructura y uso de los archivos binarios que almacenan modelos entrenados, tales como `.pkl` y `.joblib`, detallando buenas prácticas para su manejo, carga y actualización.

De manera opcional, se incluirán ejemplos introductorios con la librería **PyTorch**, orientados a quienes deseen explorar modelos más avanzados basados en redes neuronales.

1. Principales comandos de Pandas

Pandas es una librería de Python para manipulación de datos y análisis. A continuación se presentan los comandos más utilizados:

Lectura y escritura de datos

- `pd.read_csv()` - Leer archivos CSV
- `pd.read_excel()` - Leer archivos Excel
- `pd.read_json()` - Leer archivos JSON

Ejemplo de JSON:

```
{
  "nombre": "Juan",
  "edad": 30,
  "ciudad": "Madrid",
  "hobbies": ["lectura", "programación", "deportes"],
  "activo": true
}
```

- `df.to_csv()` - Guardar a CSV
- `df.to_excel()` - Guardar a Excel

Exploración y visualización de datos

- `df.head()` - Primeras filas

- `df.tail()` - Últimas filas
- `df.info()` - Información del DataFrame
- `df.describe()` - Estadísticas descriptivas
- `df.shape` - Dimensiones (filas, columnas)
- `df.columns` - Nombres de columnas
- `df.dtypes` - Tipos de datos

Selección y filtrado

- `df[columna]` - Seleccionar columna
- `df[['col1', 'col2']]` - Seleccionar múltiples columnas
- `df.loc[]` - Selección por etiquetas (A, B, C, nombres personalizados, etc.)
- `df.iloc[]` - Selección por posición (0, 1, 2, 3...)
- `df.query()` - Filtrado con expresiones

Manipulación de datos

- `df.drop()` - Eliminar filas/columnas
- `df.rename()` - Renombrar columnas
- `df.sort_values()` - Ordenar valores
- `df.groupby()` - Agrupar datos
- `df.pivot_table()` - Crear tabla dinámica
- `df.merge()` - Combinar DataFrames
- `df.concat()` - Concatenar DataFrames

Limpieza de datos

- `df.isna() / df.isnull()` - Detectar valores nulos
- `df.dropna()` - Eliminar valores nulos
- `df.fillna()` - Rellenar valores nulos
- `df.duplicated()` - Detectar duplicados
- `df.drop_duplicates()` - Eliminar duplicados

Operaciones estadísticas

- `df.mean()` - Media
- `df.median()` - Mediana
- `df.std()` - Desviación estándar
- `df.sum()` - Suma
- `df.count()` - Conteo
- `df.corr()` - Correlación

Transformaciones

- `df.apply()` - Aplicar función
- `df.map()` - Mapear valores
- `df.replace()` - Reemplazar valores

- `df.astype()` - Cambiar tipo de datos

En pandas, `df` es una convención para referirse a un DataFrame, y `pd` para referirse a la librería Pandas ya importada.

1.2 ¿Qué es `df`?

`df` es una variable que típicamente contiene un DataFrame de pandas, una estructura bidimensional con filas y columnas (similar a una hoja de Excel o una tabla SQL).

Características principales:

1. Estructura de datos: tabla con filas (observaciones) y columnas (variables/features)
2. Tipos de datos: cada columna puede tener un tipo diferente (números, texto, fechas, etc.)
3. Operaciones: panda permite manipular, filtrar, transformar y analizar datos

Ejemplo típico de uso:

```
# Cargar datos desde un archivo Excel
df = pd.read_excel('archivo.xlsx')

# Ver las primeras filas
df.head()

# Ver información del DataFrame
df.info()
```

Ver estadísticas descriptivas
`df.describe()`

En el proyecto, `df` se usa para almacenar los datos de paneles solares, y para cualquier otro proyecto indica los datos que serán cargados desde Excel. El nombre puede variar o se puede usar cualquier letra. Es recomendable no usar `i` ni `j` ya que están reservadas para números complejos.

1.3 Ejemplo simple con Pandas

A continuación se presenta un ejemplo básico y fácil de entender para operar con pandas:

In [211...]

```
# =====
# EJEMPLO SIMPLE CON PANDAS
# =====

import pandas as pd

# 1. Crear un DataFrame simple desde un diccionario
# Un diccionario es un tipo de variable que almacena datos en forma de clave-val
datos = {
    'Nombre': ['Ana', 'Luis', 'María', 'Carlos', 'Laura'],
```

```

        'Edad': [25, 30, 28, 35, 22],
        'Ciudad': ['Madrid', 'Barcelona', 'Valencia', 'Madrid', 'Sevilla'],
        'Salario': [3000, 3500, 3200, 4000, 2800]
    }

# 2. Crear el DataFrame con los datos del diccionario
df = pd.DataFrame(datos)

print("DataFrame creado:")
print(df)
print("\n" + "-" * 50)

```

DataFrame creado:

	Nombre	Edad	Ciudad	Salario
0	Ana	25	Madrid	3000
1	Luis	30	Barcelona	3500
2	Maria	28	Valencia	3200
3	Carlos	35	Madrid	4000
4	Laura	22	Sevilla	2800

In [212...]

```

# 2. Ver información básica del DataFrame
print("Primeras 3 filas:")
print(df.head(3)) # Imprime las primeras 3 filas del DataFrame

print("\nInformación del DataFrame:")
print(f"Filas: {df.shape[0]}, Columnas: {df.shape[1]}") # Imprime el número de

print("\nEstadísticas básicas:")
print(df.describe()) # Imprime las estadísticas descriptivas del DataFrame

```

Primeras 3 filas:

	Nombre	Edad	Ciudad	Salario
0	Ana	25	Madrid	3000
1	Luis	30	Barcelona	3500
2	Maria	28	Valencia	3200

Información del DataFrame:

Filas: 5, Columnas: 4

Estadísticas básicas:

	Edad	Salario
count	5.000000	5.000000
mean	28.000000	3300.000000
std	4.949747	469.041576
min	22.000000	2800.000000
25%	25.000000	3000.000000
50%	28.000000	3200.000000
75%	30.000000	3500.000000
max	35.000000	4000.000000

In [213...]

```

# 3. Seleccionar columnas
print("Solo la columna 'Nombre':")
print(df['Nombre']) # Imprime la columna 'Nombre' del DataFrame

print("\nMúltiples columnas:")
print(df[['Nombre', 'Edad', 'Salario']]) # Imprime las columnas 'Nombre', 'Edad'

```

```
Solo la columna 'Nombre':
0      Ana
1    Luis
2   María
3  Carlos
4  Laura
Name: Nombre, dtype: object
```

Múltiples columnas:

	Nombre	Edad	Salario
0	Ana	25	3000
1	Luis	30	3500
2	María	28	3200
3	Carlos	35	4000
4	Laura	22	2800

```
In [214...]: # 4. Filtrar datos
print("Personas mayores de 27 años:")
print(df[df['Edad'] > 27]) # Imprime las filas del DataFrame donde la columna 'Edad' es mayor que 27

print("\nPersonas de Madrid:")
print(df[df['Ciudad'] == 'Madrid']) # Imprime las filas del DataFrame donde la columna 'Ciudad' es igual a 'Madrid'
```

Personas mayores de 27 años:

	Nombre	Edad	Ciudad	Salario
1	Luis	30	Barcelona	3500
2	María	28	Valencia	3200
3	Carlos	35	Madrid	4000

Personas de Madrid:

	Nombre	Edad	Ciudad	Salario
0	Ana	25	Madrid	3000
3	Carlos	35	Madrid	4000

```
In [215...]: # 5. Operaciones básicas
print("Salario promedio:", df['Salario'].mean()) # Imprime el promedio del salario
print("Edad promedio:", df['Edad'].mean()) # Imprime el promedio de la edad del DataFrame
print("Salario máximo:", df['Salario'].max()) # Imprime el máximo salario del DataFrame

print("\nAgrupar por ciudad y calcular salario promedio:")
print(df.groupby('Ciudad')['Salario'].mean()) # Imprime el promedio de los salarios por ciudad
```

Salario promedio: 3300.0

Edad promedio: 28.0

Salario máximo: 4000

Agrupar por ciudad y calcular salario promedio:

Ciudad	Salario
Barcelona	3500.0
Madrid	3500.0
Sevilla	2800.0
Valencia	3200.0

Name: Salario, dtype: float64

```
In [216...]: # 6. Ordenar datos
print("Ordenado por Salario (de mayor a menor):")
print(df.sort_values('Salario', ascending=False)) # Imprime el DataFrame ordenado por salario de mayor a menor
```

Ordenado por Salario (de mayor a menor):

	Nombre	Edad	Ciudad	Salario
3	Carlos	35	Madrid	4000
1	Luis	30	Barcelona	3500
2	Maria	28	Valencia	3200
0	Ana	25	Madrid	3000
4	Laura	22	Sevilla	2800

2. ¿Qué es NumPy?

NumPy (Numerical Python) es una biblioteca fundamental de Python para computación científica y análisis numérico.

Características principales:

- Arrays multidimensionales:** Estructura de datos principal llamada `ndarray` (N-dimensional array) que permite trabajar con matrices $n \times m$ (n filas x m columnas) y vectores ($1 \times m$ vector fila, o $n \times 1$ vector columna) de manera eficiente
- Operaciones matemáticas:** Funciones optimizadas para operaciones matemáticas, algebraicas y estadísticas
- Rendimiento:** Implementado en C, es mucho más rápido que las listas de Python para operaciones numéricas
- Base para otras librerías:** Pandas, scikit-learn y muchas otras librerías de ciencia de datos están construidas sobre NumPy

Conceptos clave:

- **Array (`np.array`):** Estructura de datos principal, similar a una lista pero más eficiente. Un array es un arreglo donde se guardan los datos numéricos
- **Operaciones vectorizadas:** Permite realizar operaciones en arrays completos sin bucles, por ejemplo multiplicación de filas por columnas como en álgebra lineal, o operación punto a punto entre vectores o matrices
- **Broadcasting:** Permite operaciones entre arrays de diferentes tamaños
- **Indexación avanzada:** Acceso eficiente a elementos usando índices, máscaras booleanas, etc.

Ejemplo típico de uso:

```
import numpy as np
```

```
# Crear un array
numeros = np.array([1, 2, 3, 4, 5])

# Operaciones matemáticas
print(numeros * 2) # Multiplicar todos los elementos por 2
print(numeros.mean()) # Calcular la media
print(np.sort(numeros)) # Ordenar el array
```

En el proyecto, NumPy se usa para operaciones numéricas, cálculos matemáticos y como base para las operaciones de pandas y scikit-learn.

2.1 Ejemplo simple con NumPy

A continuación se presenta un ejemplo básico para trabajar con NumPy:

In [217...]

```
# 1. Ordenar datos con NumPy
import numpy as np

# Crear un array de ejemplo con los salarios
salarios = np.array([3000, 3500, 3200, 4000, 2800])

print("Array original de salarios:")
print(salarios)

print("\nOrdenado de menor a mayor:")
print(np.sort(salarios)) # Ordena el array de menor a mayor

print("\nOrdenado de mayor a menor (usando índices):")
indices_ordenados = np.argsort(salarios)[::-1] # Obtiene los índices ordenados
print(salarios[indices_ordenados]) # Imprime el array ordenado usando los índices

print("\nOrdenado de mayor a menor (método directo):")
print(-np.sort(-salarios)) # Ordena de mayor a menor multiplicando por -1
```

Array original de salarios:
[3000 3500 3200 4000 2800]

Ordenado de menor a mayor:
[2800 3000 3200 3500 4000]

Ordenado de mayor a menor (usando índices):
[4000 3500 3200 3000 2800]

Ordenado de mayor a menor (método directo):
[4000 3500 3200 3000 2800]

3. Principales comandos de scikit-learn (sklearn)

Scikit-learn es una librería de Python para aprendizaje automático y machine learning. A continuación se presentan los comandos más utilizados:

Preprocesamiento de datos

- `StandardScaler()` - Estandarizar características (media=0, std=1) - Ejemplo en esta guía
- `LabelEncoder()` - Codificar etiquetas categóricas a numéricas - Ejemplo en esta guía
- `SimpleImputer()` - Rellenar valores faltantes
- `train_test_split()` - Dividir datos en entrenamiento y prueba (generalmente 70% entrenamiento y 30% prueba)
- `MinMaxScaler()` - Normalizar datos a rango [0,1]
- `RobustScaler()` - Escalar usando mediana y rango intercuartil

Modelos de regresión

- `LinearRegression()` - Regresión lineal
- `Ridge()` - Regresión Ridge (regularización L2)
- `Lasso()` - Regresión Lasso (regularización L1)
- `ElasticNet()` - Regresión Elastic Net (L1 + L2)
- `DecisionTreeRegressor()` - Árbol de decisión para regresión
- `RandomForestRegressor()` - Bosque aleatorio para regresión
- `GradientBoostingRegressor()` - Gradient Boosting para regresión

Evaluación de modelos

- `mean_squared_error()` - Error cuadrático medio (MSE)
- `r2_score()` - Coeficiente de determinación R²
- `mean_absolute_error()` - Error absoluto medio (MAE)
- `cross_val_score()` - Validación cruzada
- `GridSearchCV()` - Búsqueda de hiperparámetros

Métodos principales de modelos

- `.fit()` - Entrenar el modelo
- `.predict()` - Realizar predicciones
- `.score()` - Calcular score del modelo relacionado con las métricas de evaluación
- `.get_params()` - Obtener parámetros del modelo (cuáles son las variables predictoras y su peso)
- `.set_params()` - Establecer parámetros del modelo

Transformadores

- `.fit()` - Ajustar el transformador a los datos
- `.transform()` - Transformar los datos
- `.fit_transform()` - Ajustar y transformar en un paso
- `.inverse_transform()` - Transformación inversa

Validación y selección de modelo

- `cross_val_score()` - Validación cruzada con scoring, es decir evalúa la métrica de error en cada evaluación por grupo de datos
- `RandomizedSearchCV()` - Búsqueda aleatoria de hiperparámetros
- `KFold()` - División en k-folds para validación cruzada (grupos de datos k-fold)

Modelos de clasificación

Scikit-learn también ofrece modelos especializados para problemas de clasificación:

- `LogisticRegression()` - Regresión logística (clasificación binaria y multiclas)
- `DecisionTreeClassifier()` - Árbol de decisión para clasificación
- `RandomForestClassifier()` - Bosque aleatorio para clasificación
- `GradientBoostingClassifier()` - Gradient Boosting para clasificación

- `SVC()` - Máquina de vectores de soporte (SVM) para clasificación
- `KNeighborsClassifier()` - K vecinos más cercanos (KNN) para clasificación
- `GaussianNB()` - Clasificador Bayesiano ingenuo Gaussiano

Métricas de evaluación para clasificación

Las métricas de clasificación son diferentes a las de regresión:

Métricas principales:

- `accuracy_score()` - Exactitud (proporción de predicciones correctas)
- `precision_score()` - Precisión (proporción de verdaderos positivos entre todos los positivos predichos)
- `recall_score()` - Recall/Sensibilidad (proporción de positivos detectados correctamente)
- `f1_score()` - F1-Score (media armónica de precisión y recall)
- `confusion_matrix()` - Matriz de confusión
- `classification_report()` - Reporte completo de métricas de clasificación

Diferencia clave con regresión:

- En **regresión** predecimos valores numéricos continuos (ej: precio de una casa)
- En **clasificación** predecimos categorías/clases (ej: si un cliente comprará o no, 0 o 1)

3.2 ¿Qué es StandardScaler?

StandardScaler es una herramienta de preprocessamiento de scikit-learn que estandariza las características (features) de tus datos.

¿Qué hace?

Transforma los datos para que tengan:

- **Media = 0**
- **Desviación estándar = 1**

Fórmula

Para cada valor `x`, calcula:

$$x_{\text{estandarizado}} = (x - \text{media}) / \text{desviación_estándar}$$

¿Por qué es importante?

1. **Escalas diferentes:** Si tus variables tienen rangos muy distintos (ej: edad 0-100, ingresos 0-1000000), algunos algoritmos (regresión, SVM, KNN, redes neuronales) pueden verse sesgados por las variables con valores más grandes.
2. **Convergencia más rápida:** Ayuda a que algoritmos basados en gradientes converjan más rápido.

3. **Comparación justa:** Pone todas las variables en la misma escala.

Ejemplo práctico:

```
# Datos originales
edad: [25, 30, 35, 40, 45]
ingresos: [30000, 50000, 70000, 90000, 110000]
```

```
# Despues de StandardScaler
edad: [-1.41, -0.71, 0, 0.71, 1.41]
ingresos: [-1.41, -0.71, 0, 0.71, 1.41]
```

Ambas variables quedan en la misma escala, facilitando la comparación.

Uso típico:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test) # Usa la misma media y std del entrenamiento
```

Importante: Ajusta (`fit`) solo con los datos de entrenamiento y luego transforma tanto entrenamiento como prueba con esos mismos parámetros para evitar data leakage.

¿Quieres que revise cómo lo estás usando en tu código?

3.3 ¿Qué es LabelEncoder?

Label Encoder es una herramienta de `sklearn.preprocessing` que convierte etiquetas categóricas (texto) en números. Los algoritmos de Machine Learning suelen requerir datos numéricos, así que convierte categorías en enteros.

¿Cómo funciona?

Asigna un número único a cada categoría única:

Ejemplo:

```
from sklearn.preprocessing import LabelEncoder

# Datos originales (categóricos)
colores = ['Rojo', 'Azul', 'Verde', 'Rojo', 'Azul']

# Crear y aplicar LabelEncoder
le = LabelEncoder()
colores_numericos = le.fit_transform(colores)

# Resultado:
# ['Rojo', 'Azul', 'Verde', 'Rojo', 'Azul']
# se convierte en:
# [2, 0, 1, 2, 0]
```

Características importantes:

1. Asignación automática: asigna números según el orden alfabético o de aparición
2. Reversible: puedes volver a las categorías originales con `inverse_transform()`
3. Útil para: variables categóricas nominales (sin orden inherente)

Ejemplo práctico:

Antes:

```
Ciudad: ['Madrid', 'Barcelona', 'Valencia', 'Madrid']
```

Después de LabelEncoder:

```
Ciudad: [1, 0, 2, 1] # Barcelona=0, Madrid=1, Valencia=2
```

Limitación importante:

Label Encoder asigna números que pueden interpretarse como orden ($0 < 1 < 2$). Si las categorías no tienen orden, es mejor usar **One-Hot Encoding** para evitar que el modelo asuma relaciones numéricas entre categorías.

En scikit-learn, los modelos siguen un patrón consistente de uso.

3.4 ¿Qué es scikit-learn?

scikit-learn (sklearn) es una biblioteca de Python que proporciona herramientas simples y eficientes para análisis predictivo de datos y machine learning.

Características principales:

1. **API consistente**: Todos los modelos siguen el mismo patrón (`fit`, `predict`, `score`), en este orden
2. **Preprocesamiento**: Herramientas para limpiar y preparar datos
3. **Modelos**: Algoritmos de aprendizaje supervisado y no supervisado
4. **Evaluación**: Métricas para medir el rendimiento de los modelos
5. **Validación**: Herramientas para validar y seleccionar modelos

Patrón típico de uso en algoritmos de machine learning:

```
# 1. Importar el modelo
from sklearn.linear_model import LinearRegression

# 2. Crear instancia del modelo
modelo = LinearRegression()

# 3. Entrenar el modelo (fit)
modelo.fit(X_entrenamiento, y_entrenamiento)

# 4. Hacer predicciones (predict)
predicciones = modelo.predict(X_prueba)
```

```
# 5. Evaluar el modelo (score o métricas)
```

```
r2 = modelo.score(X_prueba, y_prueba)
```

En el proyecto, sklearn se usa para entrenar modelos de regresión que predicen el consumo energético basándose en características de los paneles solares.

3.5 np.random.seed(42) — Explicación

`np.random.seed(42)` fija la semilla para la generación de números aleatorios en NumPy. En análisis de datos, esto es fundamental para garantizar la reproducibilidad de los experimentos.

¿Qué hace?

- Fija el punto de partida del generador de números aleatorios.
- Con la misma semilla, la secuencia de números "aleatorios" será la misma en cada ejecución.
- Útil para reproducibilidad: los resultados son consistentes entre ejecuciones.

¿Por qué el número 42?

Es una convención (referencia a "La guía del autoestopista galáctico"). Puede ser cualquier entero; lo importante es usar el mismo valor para reproducir resultados. El uso del número 42 como parámetro `random_state` en el aprendizaje automático es una referencia humorística a la serie de ciencia ficción "La guía del autoestopista galáctico" de Douglas Adams. En la serie, el número 42 es conocido como la "Respuesta a la Última Pregunta de la Vida, el Universo y Todo", aunque la pregunta real es desconocida. Es una referencia caprichosa que ha sido adoptada por la comunidad de programación y ciencia de datos.

Importante: No hay una razón técnica específica para usar el número 42 sobre cualquier otro valor. El propósito de establecer una semilla aleatoria (`random_state`) en aprendizaje automático es asegurar la reproducibilidad. Cuando usas la misma semilla aleatoria, obtendrás la misma secuencia de números aleatorios cada vez que ejecutes tu código. Esto es útil para depurar, compartir resultados y comparar diferentes modelos.

La comunidad de ciencia de datos suele elegir números comúnmente reconocidos como el 42 como semilla aleatoria por tradición, pero cualquier valor entero funcionará igual de bien para lograr la reproducibilidad

Ejemplo práctico:

```
import numpy as np
```

```
# Sin seed: cada ejecución da números diferentes
```

```
np.random.random(3) # Puede dar: [0.374, 0.950, 0.732]
```

```
np.random.random(3) # Puede dar: [0.598, 0.156, 0.156]
```

```
# Con seed: siempre da los mismos números
```

```
np.random.seed(42)
```

```
np.random.random(3) # Siempre da: [0.37454012, 0.95071431, 0.73199394]

np.random.seed(42) # Resetear al inicio
np.random.random(3) # De nuevo: [0.37454012, 0.95071431, 0.73199394]
```

¿Cuándo usarlo?

- En machine learning y análisis de datos: para que los experimentos sean reproducibles.
- En pruebas: para garantizar resultados consistentes.
- Al compartir código: para que otros obtengan los mismos resultados.

Resumen: Fija el generador para obtener la misma secuencia en cada ejecución, útil para reproducibilidad. El 42 es arbitrario, aunque popular.

3.6 Ejemplo simple con scikit-learn (Regresión)

A continuación se presenta un ejemplo básico y fácil de entender para operar con scikit-learn en problemas de regresión:

In [218...]

```
# =====
# EJEMPLO SIMPLE CON SCIKIT-LEARN
# =====

import pandas as pd # Para crear el DataFrame
import numpy as np # Para crear arrays
from sklearn.model_selection import train_test_split # Para dividir los datos
from sklearn.linear_model import LinearRegression # Para crear el modelo de regresión
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error # Para evaluar el modelo
from sklearn.preprocessing import StandardScaler # Para estandarizar las características
```

1. Crear datos de ejemplo. Simulamos datos de casas: tamaño (m^2) y precio usando Python.

```
np.random.seed(42) # Para reproducibilidad, seed es una semilla que genera números aleatorios.
tamaño = np.random.randint(50, 200, 20) # Tamaño en  $m^2$ 
precio = tamaño * 1500 + np.random.randint(-20000, 20000, 20) # Precio relacionado con el tamaño
```

2. Crear DataFrame usando Pandas

```
df_casas = pd.DataFrame({
    'Tamaño_m2': tamaño,
    'Precio': precio
})
```

```
Datos de ejemplo (casas):
print(df_casas.head())
print("\n" + "-" * 50) # Imprime la línea punteada
```

Datos de ejemplo (casas):

	Tamaño_m2	Precio
0	152	213311
1	142	230819
2	64	115188
3	156	231568
4	121	181269

```
In [219... # 3. Separar características (X) y variable objetivo (y)
X = df_casas[['Tamaño_m2']] # Característica: tamaño de la casa
y = df_casas['Precio'] # Variable objetivo: precio de la casa

print("Características (X):")
print(X.head())
print("\nVariable objetivo (y):")
print(y.head())
print("\n" + "-" * 50) # Imprime la línea punteada
```

Características (X):

	Tamaño_m2
0	152
1	142
2	64
3	156
4	121

Variable objetivo (y):

	Precio
0	213311
1	230819
2	115188
3	231568
4	181269

Name: Precio, dtype: int32

```
In [220... # 4. Dividir datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3, # 30% para prueba, 70% para entrenamiento
    random_state=42 # Para reproducibilidad
)

print(f"Datos de entrenamiento: {X_train.shape[0]} muestras")
print(f"Datos de prueba: {X_test.shape[0]} muestras")
print("\n" + "-" * 50) # Imprime la línea punteada
```

Datos de entrenamiento: 14 muestras

Datos de prueba: 6 muestras

```
In [221... # 5. Crear y entrenar el modelo de regresión Lineal
modelo = LinearRegression() # Crear instancia del modelo
modelo.fit(X_train, y_train) # Entrenar el modelo con datos de entrenamiento

print("Modelo entrenado exitosamente!")
print(f"Coeficiente (pendiente): {modelo.coef_[0]:.2f}") # Imprime el coeficiente
print(f"Intercepto: {modelo.intercept_:.2f}") # Imprime el intercepto del modelo
print("\n" + "-" * 50) # Imprime la línea punteada
```

Modelo entrenado exitosamente!

Coeficiente (pendiente): 1520.32

Intercepto: -7008.01

```
In [222... # 6. Hacer predicciones
y_pred = modelo.predict(X_test) # Predecir precios para datos de prueba
```

```

print("Predicciones vs Valores reales:")
resultados = pd.DataFrame({
    'Tamaño_m2': X_test['Tamaño_m2'].values,
    'Precio_Real': y_test.values,
    'Precio_Predicho': y_pred
})
print(resultados.head(10))
print("\n" + "-" * 50) # imprime la linea punteada

```

Predicciones vs Valores reales:

	Tamaño_m2	Precio_Real	Precio_Predicho
0	152	213311	224080.733746
1	137	217051	201275.923399
2	102	134899	148064.699256
3	142	230819	208877.526848
4	124	191658	181511.754431
5	70	113693	99414.437182

In [223...]

```

# 7. Evaluar el modelo con métricas
mse = mean_squared_error(y_test, y_pred) # Error cuadrático medio
r2 = r2_score(y_test, y_pred) # Coeficiente de determinación R²
mae = mean_absolute_error(y_test, y_pred) # Error absoluto medio

print("Métricas de evaluación del modelo:")
print(f"Error Cuadrático Medio (MSE): {mse:.2f}") # Imprime el error cuadrático
print(f"Coeficiente de Determinación (R²): {r2:.4f}") # Imprime el R² (1.0 es p
print(f"Error Absoluto Medio (MAE): {mae:.2f}") # Imprime el error absoluto mea
print("\n" + "-" * 50) # Imprime la línea punteada

```

Métricas de evaluación del modelo:

Error Cuadrático Medio (MSE): 221071290.50
 Coeficiente de Determinación (R²): 0.8852
 Error Absoluto Medio (MAE): 14346.13

In [224...]

```

# 8. Hacer una predicción para una nueva casa
nueva_casa = pd.DataFrame({'Tamaño_m2': [100]}) # Casa de 100 m²
precio_predicho = modelo.predict(nueva_casa) # Predecir el precio

print(f"Para una casa de {nueva_casa['Tamaño_m2'][0]} m²:")
print(f"Precio predicho: ${precio_predicho[0]:,.2f}") # Imprime el precio predi

```

Para una casa de 100 m²:
 Precio predicho: \$145,024.06

3.7 Ejemplo simple con scikit-learn - Clasificación con Random Forest

A continuación se presenta un ejemplo básico y fácil de entender para operar con scikit-learn en clasificación usando Random Forest:

In [225...]

```

# =====
# EJEMPLO SIMPLE CON SCIKIT-LEARN - CLASIFICACIÓN CON RANDOM FOREST
# =====

```

```

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.preprocessing import LabelEncoder

# 1. Crear datos de ejemplo. Simulamos datos de clientes: edad, ingresos y si compró
np.random.seed(42) # Para reproducibilidad
n_samples = 200

# Generar datos sintéticos
edad = np.random.randint(18, 70, n_samples) # Edad entre 18 y 70 años
ingresos = np.random.randint(20000, 100000, n_samples) # Ingresos entre 20k y 100k

# Crear variable objetivo (si compró o no) basada en edad e ingresos
# Lógica: mayores ingresos y edad entre 30-50 años tienen mayor probabilidad de compra
probabilidad_compra = (ingresos / 50000) + (edad >= 30) * (edad <= 50) * 0.5
compra = (probabilidad_compra + np.random.normal(0, 0.3, n_samples)) > 1.5).astype(int)

# 2. Crear DataFrame usando Pandas
df_clientes = pd.DataFrame({
    'Edad': edad,
    'Ingresos': ingresos,
    'Compro': compra # 1 = Sí compró, 0 = No compró
})

print("Datos de ejemplo (clientes):")
print(df_clientes.head(10))
print("\nDistribución de compras:")
print(df_clientes['Compro'].value_counts())
print("\n" + "-" * 50)

```

Datos de ejemplo (clientes):

	Edad	Ingresos	Compro
0	56	78053	1
1	69	41959	0
2	46	25530	0
3	32	23748	0
4	60	33545	0
5	25	86199	1
6	38	54766	1
7	56	93530	1
8	36	81087	1
9	40	88840	1

Distribución de compras:

Compro	count
0	114
1	86

Name: count, dtype: int64

In [226...]

```

# 3. Separar características (X) y variable objetivo (y)
X = df_clientes[['Edad', 'Ingresos']] # Características: edad e ingresos
y = df_clientes['Compro'] # Variable objetivo: si compró (0 o 1)

print("Características (X):")
print(X.head())

```

```

print("\nVariable objetivo (y):")
print(y.head())
print(f"\nForma de X: {X.shape}") # (200, 2) - 200 muestras, 2 características
print(f"Forma de y: {y.shape}") # (200,) - 200 muestras
print("\n" + "-" * 50)

```

Características (X):

	Edad	Ingresos
0	56	78053
1	69	41959
2	46	25530
3	32	23748
4	60	33545

Variable objetivo (y):

0	1
1	0
2	0
3	0
4	0

Name: Compro, dtype: int64

Forma de X: (200, 2)

Forma de y: (200,)

In [227...]

```

# 4. Dividir datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.3, # 30% para prueba, 70% para entrenamiento
    random_state=42, # Para reproducibilidad
    stratify=y # Mantiene la proporción de clases en train y test
)

print(f"Datos de entrenamiento: {X_train.shape[0]} muestras")
print(f"Datos de prueba: {X_test.shape[0]} muestras")
print("\nDistribución en entrenamiento:")
print(y_train.value_counts())
print("\nDistribución en prueba:")
print(y_test.value_counts())
print("\n" + "-" * 50)

```

Datos de entrenamiento: 140 muestras

Datos de prueba: 60 muestras

Distribución en entrenamiento:

Compro	
0	80
1	60

Name: count, dtype: int64

Distribución en prueba:

Compro	
0	34
1	26

Name: count, dtype: int64

```
In [228...]: # 5. Crear y entrenar el modelo Random Forest
# Random Forest combina múltiples árboles de decisión para mejorar la precisión
modelo = RandomForestClassifier(
    n_estimators=100,      # Número de árboles en el bosque
    random_state=42,       # Para reproducibilidad
    max_depth=5            # Profundidad máxima de cada árbol
)

modelo.fit(X_train, y_train) # Entrenar el modelo con datos de entrenamiento

print("Modelo Random Forest entrenado exitosamente!")
print(f"Número de árboles: {modelo.n_estimators}")
print(f"Importancia de características:")
for i, feature in enumerate(X.columns):
    print(f"  {feature}: {modelo.feature_importances_[i]:.4f}")
print("\n" + "-" * 50)
```

Modelo Random Forest entrenado exitosamente!

Número de árboles: 100

Importancia de características:

Edad: 0.2599

Ingresos: 0.7401

```
In [229...]: # 6. Hacer predicciones
y_pred = modelo.predict(X_test) # Predecir clases (0 o 1) para datos de prueba
y_pred_proba = modelo.predict_proba(X_test) # Predecir probabilidades

print("Predicciones vs Valores reales:")
resultados = pd.DataFrame({
    'Edad': X_test['Edad'].values,
    'Ingresos': X_test['Ingresos'].values,
    'Compro_Real': y_test.values,
    'Compro_Predicho': y_pred,
    'Probabilidad_Comprar': y_pred_proba[:, 1] # Probabilidad de comprar (clase 1)
})
print(resultados.head(10))
print("\n" + "-" * 50)
```

Predicciones vs Valores reales:

	Edad	Ingresos	Compro_Real	Compro_Predicho	Probabilidad_Comprar
0	32	58467	1	0	0.314456
1	62	98069	1	1	0.883942
2	24	30647	0	0	0.034596
3	35	62642	1	1	0.509683
4	19	49855	0	0	0.037645
5	36	81087	1	1	0.854093
6	28	39830	0	0	0.065001
7	41	93744	1	1	0.980227
8	23	81434	1	1	0.576695
9	56	23051	0	0	0.029849

```
In [230...]: # 7. Evaluar el modelo con métricas de clasificación
accuracy = accuracy_score(y_test, y_pred) # Exactitud (proporción de predicción correcta)
precision = precision_score(y_test, y_pred) # Precisión (verdaderos positivos / predicciones positivas)
recall = recall_score(y_test, y_pred) # Recall/Sensibilidad (verdaderos positivos / todos los positivos)
f1 = f1_score(y_test, y_pred) # F1-Score (media armónica de precisión y recall)
```

```

matriz_confusion = confusion_matrix(y_test, y_pred) # Matriz de confusión

print("Métricas de evaluación del modelo:")
print(f"Exactitud (Accuracy): {accuracy:.4f} ({accuracy*100:.2f}%)")
print(f"Precisión (Precision): {precision:.4f}")
print(f"Recall/Sensibilidad: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
print("\nMatriz de confusión:")
print("                  Predicho")
print("          No(0)    Sí(1)")
print(f"Real    No(0)    {matriz_confusion[0][0]:3d}    {matriz_confusion[0][1]}")
print(f"          Sí(1)    {matriz_confusion[1][0]:3d}    {matriz_confusion[1][1]}")
print("\n" + "-" * 50)

```

Métricas de evaluación del modelo:
 Exactitud (Accuracy): 0.8000 (80.00%)
 Precisión (Precision): 0.8182
 Recall/Sensibilidad: 0.6923
 F1-Score: 0.7500

Matriz de confusión:

		Predicho	
		No(0)	Sí(1)
Real	No(0)	30	4
	Sí(1)	8	18

In [231...]: # 8. Reporte completo de clasificación
 print("Reporte completo de clasificación:")
 print(classification_report(y_test, y_pred, target_names=['No Compró', 'Compró'])
 print("-" * 50)

Reporte completo de clasificación:

	precision	recall	f1-score	support
No Compró	0.79	0.88	0.83	34
Compró	0.82	0.69	0.75	26
accuracy			0.80	60
macro avg	0.80	0.79	0.79	60
weighted avg	0.80	0.80	0.80	60

In [232...]: # 9. Hacer predicción para un nuevo cliente
 nuevo_cliente = pd.DataFrame({
 'Edad': [35],
 'Ingresos': [60000]
 })

 # Predecir clase y probabilidad
 prediccion = modelo.predict(nuevo_cliente)[0]
 probabilidad = modelo.predict_proba(nuevo_cliente)[0]

 print(f"Para un cliente de {nuevo_cliente['Edad'][0]} años con ingresos de \${nue
 print(f"Predicción: {'Sí comprará' if prediccion == 1 else 'No comprará'}")
 print(f"Probabilidad de comprar: {probabilidad[1]*100:.2f}%")
 print(f"Probabilidad de no comprar: {probabilidad[0]*100:.2f}%")
 print("\n" + "-" * 50)

Para un cliente de 35 años con ingresos de \$60,000:

Predicción: Sí comprará

Probabilidad de comprar: 52.31%

Probabilidad de no comprar: 47.69%

In [233...]

```
# 10. Comparar con el score del modelo (método directo)
score_train = modelo.score(X_train, y_train) # Exactitud en entrenamiento
score_test = modelo.score(X_test, y_test) # Exactitud en prueba

print("Score del modelo:")
print(f"Exactitud en entrenamiento: {score_train:.4f} ({score_train*100:.2f}%)")
print(f"Exactitud en prueba: {score_test:.4f} ({score_test*100:.2f}%)")

# Si hay mucha diferencia entre train y test, el modelo podría estar sobreajustado
diferencia = abs(score_train - score_test)
if diferencia > 0.1:
    print(f"\n⚠️ Atención: Diferencia alta entre entrenamiento y prueba ({diferencia:.4f})\n    El modelo podría estar sobreajustado.")
else:
    print(f"\n✓ Diferencia aceptable entre entrenamiento y prueba ({diferencia:.4f})\n    \n    + =" * 50)
```

Score del modelo:

Exactitud en entrenamiento: 0.9286 (92.86%)

Exactitud en prueba: 0.8000 (80.00%)

⚠️ Atención: Diferencia alta entre entrenamiento y prueba (0.1286)
El modelo podría estar sobreajustado.

4. Principales comandos de openpyxl

Openpyxl es una librería de Python para leer y escribir archivos Excel (.xlsx, .xlsm). A continuación se presentan los comandos más utilizados:

Abrir y cerrar archivos Excel

- `load_workbook()` - Cargar un archivo Excel existente
- `Workbook()` - Crear un nuevo archivo Excel en blanco
- `.save()` - Guardar el archivo Excel
- `.close()` - Cerrar el archivo Excel

Navegación entre hojas

- `.sheetnames` - Obtener lista de nombres de hojas
- `wb['NombreHoja']` - Seleccionar una hoja por nombre
- `wb.active` - Acceder a la hoja activa
- `wb.worksheets` - Lista de todas las hojas del libro

Lectura de datos

- `.cell(fila, columna)` - Acceder a una celda específica (fila, columna)
- `.cell(fila, columna).value` - Obtener el valor de una celda
- `.max_row` - Obtener el número de la última fila con datos
- `.max_column` - Obtener el número de la última columna con datos
- `.iter_rows()` - Iterar sobre filas
- `.iter_cols()` - Iterar sobre columnas
- `[fila][columna]` - Acceso directo por índice (ej: `ws['A1']`)

Escritura de datos

- `.cell(fila, columna, valor)` - Escribir valor en una celda
- `ws['A1'] = valor` - Asignar valor directamente a celda
- `.append()` - Agregar una fila al final de la hoja
- `.merge_cells()` - Combinar celdas
- `.unmerge_cells()` - Descombinar celdas

Formato de celdas

- `.font` - Configurar fuente (tamaño, negrita, color)
- `.fill` - Configurar relleno de celda
- `.border` - Configurar bordes
- `.alignment` - Configurar alineación (centrado, izquierda, derecha)
- `.number_format` - Configurar formato de números

Manipulación de hojas

- `.create_sheet()` - Crear una nueva hoja
- `.remove()` - Eliminar una hoja
- `.copy_worksheet()` - Copiar una hoja
- `.title` - Cambiar el nombre de la hoja

Filtros y validación

- `.auto_filter` - Aplicar filtros automáticos
- `.freeze_panes` - Congelar paneles (filas/columnas)
- `.data_validation` - Agregar validación de datos

4.2 ¿Qué es openpyxl?

openpyxl es una biblioteca de Python para leer y escribir archivos Excel (.xlsx, .xlsm).

Características principales:

1. **Lectura y escritura:** Permite leer datos de archivos Excel existentes y crear nuevos archivos Excel
2. **Formato completo:** Soporta lectura y escritura de formatos, estilos, fórmulas, gráficos, etc.

3. **Manipulación de hojas:** Permite crear, eliminar, copiar y modificar hojas de cálculo
4. **Compatibilidad:** Funciona con archivos Excel modernos (.xlsx, .xlsm), no con formatos antiguos (.xls)

Conceptos clave:

- **Workbook (wb):** Representa todo el archivo Excel (libro de trabajo)
- **Worksheet (ws):** Representa una hoja individual dentro del libro
- **Cell:** Representa una celda individual con su valor, formato, etc.
- **Notación:** Las filas se numeran desde 1, las columnas pueden ser números (1, 2, 3) o letras ('A', 'B', 'C')

Ejemplo típico de uso:

```
from openpyxl import load_workbook

# Cargar un archivo Excel existente
wb = load_workbook('archivo.xlsx')

# Seleccionar una hoja
ws = wb['Hoja1'] # o wb.active para la hoja activa

# Leer el valor de una celda
valor = ws['A1'].value # o ws.cell(1, 1).value

# Escribir en una celda
ws['A1'] = 'Nuevo valor'

# Guardar el archivo
wb.save('archivo.xlsx')
wb.close()
```

En el proyecto, openpyxl se usa cuando necesitas leer datos directamente desde archivos Excel con más control que `pd.read_excel()`, o cuando necesitas crear o modificar archivos Excel con formato específico.

4.3 Ejemplo simple con openpyxl

A continuación se presenta un ejemplo básico y fácil de entender para operar con openpyxl:

```
In [234...]: # =====
# EJEMPLO SIMPLE CON OPENPYXL
# =====

from openpyxl import Workbook, load_workbook
from openpyxl.styles import Font, Alignment, PatternFill
from openpyxl.utils import get_column_letter

# 1. Crear un nuevo archivo Excel
wb = Workbook() # Crea un nuevo Libro de trabajo
ws = wb.active # Selecciona La hoja activa (por defecto se llama "Sheet")
ws.title = "Datos" # Cambia el nombre de la hoja a "Datos"
```

```

print("Archivo Excel creado exitosamente")
print(f"Nombre de la hoja: {ws.title}")
print("\n" + "-" * 50) # Imprime la línea punteada

```

Archivo Excel creado exitosamente
Nombre de la hoja: Datos

In [235...]

```

# 2. Escribir datos en celdas
ws['A1'] = 'Nombre' # Escribe en la celda A1
ws['B1'] = 'Edad' # Escribe en la celda B1
ws['C1'] = 'Ciudad' # Escribe en la celda C1

# Escribir datos usando el método cell()
ws.cell(2, 1, 'Ana') # Fila 2, Columna 1 (A2)
ws.cell(2, 2, 25) # Fila 2, Columna 2 (B2)
ws.cell(2, 3, 'Madrid') # Fila 2, Columna 3 (C2)

# Agregar más filas usando append()
ws.append(['Luis', 30, 'Barcelona']) # Agrega una fila al final
ws.append(['María', 28, 'Valencia'])
ws.append(['Carlos', 35, 'Madrid'])

print("Datos escritos en el archivo Excel")
print("\n" + "-" * 50) # Imprime la línea punteada

```

Datos escritos en el archivo Excel

In [236...]

```

# 3. Leer datos de celdas
print("Lectura de datos del archivo Excel:")
print(f"Celda A1: {ws['A1'].value}") # Lee el valor de la celda A1
print(f"Celda B2: {ws.cell(2, 2).value}") # Lee el valor de la fila 2, columna

print(f"\nÚltima fila con datos: {ws.max_row}") # Imprime el número de la última fila con datos
print(f"Última columna con datos: {ws.max_column}") # Imprime el número de la última columna con datos
print("\n" + "-" * 50) # Imprime la línea punteada

```

Lectura de datos del archivo Excel:

Celda A1: Nombre

Celda B2: 25

Última fila con datos: 5

Última columna con datos: 3

In [237...]

```

# 4. Aplicar formato a las celdas (encabezados)
header_fill = PatternFill(start_color="366092", end_color="366092", fill_type="solid")
header_font = Font(bold=True, color="FFFFFF", size=12) # Texto en negrita, color blanco
header_alignment = Alignment(horizontal="center", vertical="center") # Texto centrado

# Aplicar formato a la primera fila (encabezados)
for cell in ws[1]: # Itera sobre todas las celdas de la fila 1
    cell.fill = header_fill # Aplica el color de fondo
    cell.font = header_font # Aplica la fuente
    cell.alignment = header_alignment # Aplica la alineación

```

```
print("Formato aplicado a los encabezados")
print("\n" + "-" * 50) # imprime La Linea punteada
```

Formato aplicado a los encabezados

In [238...]

```
# 5. Guardar el archivo Excel
wb.save('ejemplo_openpyxl.xlsx') # Guarda el archivo Excel con el nombre especificado
print("Archivo Excel guardado como 'ejemplo_openpyxl.xlsx'")

# Cerrar el archivo (opcional, se cierra automáticamente al terminar)
wb.close()
print("Archivo Excel cerrado")
print("\n" + "-" * 50) # Imprime La Línea punteada
```

Archivo Excel guardado como 'ejemplo_openpyxl.xlsx'

Archivo Excel cerrado

In [239...]

```
# 6. Leer un archivo Excel existente
try:
    # Cargar el archivo que acabamos de crear
    wb_lectura = load_workbook('ejemplo_openpyxl.xlsx') # Carga el archivo Excel
    ws_lectura = wb_lectura.active # Selecciona la hoja activa

    print("Lectura de archivo Excel existente:")
    print(f"Nombre de la hoja: {ws_lectura.title}") # Imprime el nombre de la hoja
    print(f"Número de filas: {ws_lectura.max_row}") # Imprime el número de filas
    print(f"Número de columnas: {ws_lectura.max_column}") # Imprime el número de columnas

    print("\nDatos de todas las filas:")
    # Iterar sobre todas las filas con datos
    for row in ws_lectura.iter_rows(values_only=True): # Itera sobre filas, devolviendo tuplas
        print(row) # Imprime cada fila

    wb_lectura.close() # Cierra el archivo

except FileNotFoundError:
    print("El archivo no existe aún. Ejecuta primero las celdas anteriores.")
```

Lectura de archivo Excel existente:

Nombre de la hoja: Datos

Número de filas: 5

Número de columnas: 3

Datos de todas las filas:

```
('Nombre', 'Edad', 'Ciudad')
('Ana', 25, 'Madrid')
('Luis', 30, 'Barcelona')
('María', 28, 'Valencia')
('Carlos', 35, 'Madrid')
```

5. ¿Qué son los modelos PKL (Pickle) y por qué se guardan?

Los archivos `.pk1` (Pickle) son archivos binarios de Python que permiten **serializar** (convertir a bytes) y **deserializar** (recuperar) objetos de Python para guardarlos en disco

y reutilizarlos después. Los bytes son representaciones binarias compuestas de ceros (0) y unos (1).

5.1 ¿Qué es Pickle?

Pickle es un módulo de Python que permite guardar objetos de Python (como modelos de machine learning entrenados) en archivos binarios (`.pk1`). Esto es útil porque:

1. **Preservación del estado:** Guarda el modelo completo con todos sus parámetros entrenados
2. **Reutilización:** Permite usar el modelo sin tener que entrenarlo de nuevo
3. **Persistencia:** El modelo se guarda permanentemente en disco
4. **Eficiencia:** Evita re-entrenar modelos que pueden tardar horas o días

¿Por qué se guardan los modelos?

Después de entrenar un modelo de machine learning, normalmente:

- El entrenamiento puede tomar mucho tiempo (minutos, horas o días)
- El modelo contiene parámetros aprendidos (pesos, coeficientes, árboles, etc.)
- Necesitas usar el modelo en el futuro sin volver a entrenarlo
- Quieres compartir el modelo con otros usuarios o sistemas

Sin guardar el modelo: Tendrías que entrenarlo cada vez que lo necesites (muy ineficiente)

Con el modelo guardado: Solo cargas el archivo `.pk1` y haces predicciones inmediatamente. La idea es que el despliegue de este modelo se realice en Excel, lo cual se aprenderá a hacer en este curso.

5.2 ¿Qué contiene un modelo PKL?

Un archivo `.pk1` puede contener:

1. **El modelo entrenado:** Con todos sus parámetros y pesos aprendidos

2. **Objetos auxiliares:**

- `StandardScaler` (escalador de datos)
- `LabelEncoder` (codificador de etiquetas)
- `SimpleImputer` (para valores faltantes)
- Lista de nombres de características (features)
- Metadatos del modelo

3. **Información adicional:**

- Métricas de evaluación (R^2 , RMSE, MAE)
- Parámetros del modelo
- Fecha de entrenamiento

5.3 Alternativas a Pickle

- **Joblib:** Similar a Pickle pero más eficiente para arrays grandes de NumPy (recomendado para scikit-learn)
- **HDF5:** Para modelos grandes y datasets
- **ONNX:** Formato estándar para intercambiar modelos entre frameworks
- **TensorFlow SavedModel / PyTorch:** Formatos específicos de estos frameworks

5.4 ¿Qué es un Framework?

Un **framework** (marco de trabajo) es un conjunto de herramientas, bibliotecas y reglas que facilita el desarrollo de software. Proporciona una estructura base y funciones comunes para que te enfoques en la lógica de tu aplicación.

Analogía simple

Si construir una aplicación fuera construir una casa:

- **Sin framework:** Compilas cada herramienta desde cero (martillo, tornillos, diseño).
- **Con framework:** Ya tienes herramientas y un plano base; te enfocas en los detalles.

Características principales

1. **Estructura predefinida:** Define cómo organizar tu código.
2. **Funcionalidades comunes:** Incluye funciones listas (autenticación, manejo de datos, etc.).
3. **Patrones de diseño:** Usa buenas prácticas ya probadas.
4. **Ahorro de tiempo:** Evita escribir código repetitivo.

Ejemplos comunes de Framework

- **Para Machine Learning:** TensorFlow, PyTorch, scikit-learn
- **Para web:** Django (Python), Flask (Python), React (JavaScript)
- **Para datos:** pandas, NumPy

Framework vs. Librería

- **Librería:** Herramientas que eliges usar cuando quieras (ej: `pandas`).
- **Framework:** Estructura que sigue tu código; tu código "llena" el framework. Por ejemplo, en machine learning con scikit-learn la estructura que sigue el código es: `fit`, `predict` y finalmente `score`.

En resumen: un framework da estructura y herramientas para desarrollar más rápido y con mejores prácticas.

5.5 Ejemplo práctico: Guardar y cargar modelos PKL

A continuación se presenta un ejemplo completo y fácil de entender para operar con modelos PKL:

```
In [240...]
# =====
# EJEMPLO COMPLETO: GUARDAR Y CARGAR MODELOS PKL
# =====

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score, mean_squared_error
import pickle # Librería para guardar y cargar modelos PKL
import joblib # Alternativa más eficiente para modelos de scikit-Learn

print("=" * 70)
print("EJEMPLO: GUARDAR Y CARGAR MODELOS PKL")
print("=" * 70)

# 1. Crear datos de ejemplo
np.random.seed(42)
tamaño = np.random.randint(50, 200, 20)
precio = tamaño * 1500 + np.random.randint(-20000, 20000, 20)

df_casas = pd.DataFrame({
    'Tamaño_m2': tamaño,
    'Precio': precio
})

print("\n1. Datos de ejemplo creados:")
print(df_casas.head())
print("\n" + "-" * 70)
```

```
=====
EJEMPLO: GUARDAR Y CARGAR MODELOS PKL
=====
```

1. Datos de ejemplo creados:

	Tamaño_m2	Precio
0	152	213311
1	142	230819
2	64	115188
3	156	231568
4	121	181269

```
In [241...]
# 2. Preparar datos y entrenar modelo
X = df_casas[['Tamaño_m2']]
y = df_casas['Precio']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_

# Crear y entrenar escalador
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# Crear y entrenar modelo
modelo = LinearRegression()
modelo.fit(X_train_scaled, y_train)

# Evaluar modelo
y_pred = modelo.predict(X_test_scaled)
r2 = r2_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print("2. Modelo entrenado exitosamente!")
print(f"    R² Score: {r2:.4f}")
print(f"    RMSE: {rmse:.2f}")
print(f"    Coeficiente: {modelo.coef_[0]:.2f}")
print(f"    Intercepto: {modelo.intercept_:.2f}")
print("\n" + "-" * 70)
```

2. Modelo entrenado exitosamente!

R² Score: 0.8852
RMSE: 14868.47
Coeficiente: 65218.56
Intercepto: 206379.86

In [242...]

```
# 3. GUARDAR MODELO CON PICKLE (método estándar)
print("3. GUARDANDO MODELO CON PICKLE:")
print("    " + "-" * 66)

# Guardar solo el modelo
with open('modelo_casas.pkl', 'wb') as f: # 'wb' = write binary (escribir binario)
    pickle.dump(modelo, f)
print("    ✓ Modelo guardado: modelo_casas.pkl")

# Guardar el escalador (importante para futuras predicciones), ya que los datos
with open('scaler_casas.pkl', 'wb') as f:
    pickle.dump(scaler, f)
print("    ✓ Escalador guardado: scaler_casas.pkl")

# Guardar TODO junto (modelo + escalador + información)
modelo_completo = {
    'modelo': modelo,
    'scaler': scaler,
    'r2_score': r2,
    'rmse': rmse,
    'coeficiente': modelo.coef_[0],
    'intercepto': modelo.intercept_,
    'feature_names': ['Tamaño_m2']
}

with open('modelo_completo_casas.pkl', 'wb') as f:
    pickle.dump(modelo_completo, f)
print("    ✓ Modelo completo guardado: modelo_completo_casas.pkl")
print("\n" + "-" * 70)
```

3. GUARDANDO MODELO CON PICKLE:

```
-----  
✓ Modelo guardado: modelo_casas.pkl  
✓ Escalador guardado: scaler_casas.pkl  
✓ Modelo completo guardado: modelo_completo_casas.pkl  
-----
```

In [243...]

```
# 4. GUARDAR CON JOBLIB (método recomendado para scikit-Learn - más rápido)  
print("4. GUARDANDO MODELO CON JOBLIB (RECOMENDADO):")  
print(" " + "-" * 66)  
  
# Joblib es más eficiente que Pickle para modelos con arrays grandes de NumPy  
joblib.dump(modelo, 'modelo_casas.joblib')  
print(" ✓ Modelo guardado: modelo_casas.joblib")  
  
joblib.dump(scaler, 'scaler_casas.joblib')  
print(" ✓ Escalador guardado: scaler_casas.joblib")  
  
# Guardar todo junto con joblib  
joblib.dump(modelo_completo, 'modelo_completo_casas.joblib')  
print(" ✓ Modelo completo guardado: modelo_completo_casas.joblib")  
print("\n" + "-" * 70)
```

4. GUARDANDO MODELO CON JOBLIB (RECOMENDADO):

```
-----  
✓ Modelo guardado: modelo_casas.joblib  
✓ Escalador guardado: scaler_casas.joblib  
✓ Modelo completo guardado: modelo_completo_casas.joblib  
-----
```

In [244...]

```
# 5. CARGAR Y USAR MODELO GUARDADO (Simulando nueva sesión)  
print("5. CARGANDO MODELO DESDE ARCHIVO PKL:")  
print(" " + "-" * 66)  
  
# Simular que estamos en una nueva sesión (no tenemos el modelo en memoria)  
# Borrar variables para simular nueva sesión  
del modelo, scaler, modelo_completo  
  
# Cargar modelo con Pickle  
with open('modelo_casas.pkl', 'rb') as f: # 'rb' = read binary (Leer binario)  
    modelo_cargado = pickle.load(f)  
  
with open('scaler_casas.pkl', 'rb') as f:  
    scaler_cargado = pickle.load(f)  
  
print(" ✓ Modelo cargado desde: modelo_casas.pkl")  
print(" ✓ Escalador cargado desde: scaler_casas.pkl")  
print("\n" + "-" * 70)
```

5. CARGANDO MODELO DESDE ARCHIVO PKL:

```
-----  
✓ Modelo cargado desde: modelo_casas.pkl  
✓ Escalador cargado desde: scaler_casas.pkl  
-----
```

In [245...]

```
# 6. HACER PREDICCIONES CON EL MODELO CARGADO  
print("6. HACIENDO PREDICCIONES CON EL MODELO CARGADO:")  
print(" " + "-" * 66)
```

```
# Nueva casa para predecir
nueva_casa = pd.DataFrame({'Tamaño_m2': [100]}) # Casa de 100 m²

# IMPORTANTE: Aplicar el mismo preprocessamiento que en el entrenamiento
# 1. Escalar los datos nuevos con el escalador guardado
nueva_casa_scaled = scaler_cargado.transform(nueva_casa)

# 2. Hacer predicción
precio_predicho = modelo_cargado.predict(nueva_casa_scaled)

print(f"    Para una casa de {nueva_casa['Tamaño_m2'][0]} m²:")
print(f"    Precio predicho: ${precio_predicho[0]:,.2f}")
print("\n" + "-" * 70)
```

6. HACIENDO PREDICCIONES CON EL MODELO CARGADO:

Para una casa de 100 m²:
 Precio predicho: \$145,024.06

In [246...]

```
# 7. CARGAR MODELO COMPLETO (con toda la información)
print("7. CARGANDO MODELO COMPLETO:")
print("    " + "-" * 66)

with open('modelo_completo_casas.pkl', 'rb') as f:
    modelo_completo_cargado = pickle.load(f)

print("    Información del modelo cargado:")
print(f"        - R² Score: {modelo_completo_cargado['r2_score']:.4f}")
print(f"        - RMSE: {modelo_completo_cargado['rmse']:.2f}")
print(f"        - Coeficiente: {modelo_completo_cargado['coeficiente']:.2f}")
print(f"        - Intercepto: {modelo_completo_cargado['intercepto']:.2f}")
print(f"        - Features: {modelo_completo_cargado['feature_names']}")

# Usar el modelo del diccionario
modelo_del_diccionario = modelo_completo_cargado['modelo']
scaler_del_diccionario = modelo_completo_cargado['scaler']

# Hacer predicción
nueva_casa_scaled = scaler_del_diccionario.transform(nueva_casa)
precio_predicho = modelo_del_diccionario.predict(nueva_casa_scaled)
print(f"\n    Predicción con modelo del diccionario: ${precio_predicho[0]:,.2f}")
print("\n" + "-" * 70)
```

7. CARGANDO MODELO COMPLETO:

Información del modelo cargado:
 - R² Score: 0.8852
 - RMSE: 14868.47
 - Coeficiente: 65218.56
 - Intercepto: 206379.86
 - Features: ['Tamaño_m2']

Predicción con modelo del diccionario: \$145,024.06

In [247...]

```
# 8. CARGAR CON JOBLIB (método más rápido)
print("8. CARGANDO MODELO CON JOBLIB:")
```

```

print(" " + "-" * 66)

# Joblib es más rápido que Pickle para cargar
modelo_joblib = joblib.load('modelo_casas.joblib')
scaler_joblib = joblib.load('scaler_casas.joblib')

print(" ✓ Modelo cargado con Joblib")
print(" ✓ Escalador cargado con Joblib")

# Predicción
nueva_casa_scaled = scaler_joblib.transform(nueva_casa)
precio_predicho = modelo_joblib.predict(nueva_casa_scaled)
print(f" Predicción: ${precio_predicho[0]:,.2f}")
print("\n" + "=" * 70)

```

8. CARGANDO MODELO CON JOBLIB:

 ✓ Modelo cargado con Joblib
 ✓ Escalador cargado con Joblib
 Predicción: \$145,024.06

=====

5.6 Resumen: Flujo completo de guardado y carga

ENTRENAMIENTO:

1. Entrenar modelo → modelo.fit(X_train, y_train)
2. Guardar modelo → pickle.dump(modelo, archivo) o joblib.dump()
3. Guardar escalador → pickle.dump(scaler, archivo)

DEPLOYMENT (Despliegue):

1. Cargar modelo → modelo = pickle.load(archivo)
2. Cargar escalador → scaler = pickle.load(archivo)
3. Preprocesar datos nuevos → datos_scaled = scaler.transform(datos_nuevos)
4. Predecir → predicciones = modelo.predict(datos_scaled)

5.7 Ejemplo de despliegue en producción

A continuación se presenta un ejemplo de cómo crear una función de predicción lista para producción:

In [248...]

```

# =====
# EJEMPLO: FUNCIÓN DE PREDICCIÓN PARA PRODUCCIÓN
# =====

def predecir_precio_casa(tamaño_m2, ruta_modelo='modelo_completo_casas.pkl'):
    """
        Función para predecir el precio de una casa basándose en su tamaño.

    Parámetros:
    -----
    tamaño_m2 : float o int
        Tamaño de la casa en metros cuadrados
    ruta_modelo : str
        Ruta al archivo del modelo guardado (por defecto 'modelo_completo_casas.

```

```

Retorna:
-----
dict : Diccionario con la predicción y metadatos
"""

try:
    # Cargar modelo completo
    with open(ruta_modelo, 'rb') as f:
        modelo_completo = pickle.load(f)

    # Extraer componentes
    modelo = modelo_completo['modelo']
    scaler = modelo_completo['scaler']

    # Preparar datos
    datos_nuevos = pd.DataFrame({'Tamaño_m2': [tamaño_m2]})

    # Aplicar preprocessamiento (ESCALADO)
    datos_scaled = scaler.transform(datos_nuevos)

    # Predecir
    precio_predicho = modelo.predict(datos_scaled)[0]

    # Retornar resultado
    resultado = {
        'tamaño_m2': tamaño_m2,
        'precio_predicho': round(precio_predicho, 2),
        'modelo_usado': 'Linear Regression',
        'r2_score': modelo_completo['r2_score'],
        'rmse': modelo_completo['rmse'],
        'status': 'success'
    }

    return resultado

except FileNotFoundError:
    return {
        'status': 'error',
        'mensaje': f'No se encontró el archivo del modelo: {ruta_modelo}'
    }
except Exception as e:
    return {
        'status': 'error',
        'mensaje': f'Error al hacer predicción: {str(e)}'
    }

# Probar la función de predicción
print("=" * 70)
print("EJEMPLO: FUNCIÓN DE PREDICCIÓN PARA PRODUCCIÓN")
print("=" * 70)

# Hacer varias predicciones
casas_ejemplo = [80, 100, 120, 150, 180]

print("\nPredicciones de precios para diferentes tamaños:")
print("-" * 70)

for tamaño in casas_ejemplo:
    resultado = predecir_precio_casa(tamaño)

```

```

if resultado['status'] == 'success':
    print(f" Casa de {resultado['tamaño_m2']}:3d m² → Precio: ${resultado['precio']}")
else:
    print(f" Error: {resultado['mensaje']}")

print("\n" + "=" * 70)

```

=====
EJEMPLO: FUNCIÓN DE PREDICCIÓN PARA PRODUCCIÓN
=====

Predicciones de precios para diferentes tamaños:

Casa de 80 m² → Precio: \$114,617.64
Casa de 100 m² → Precio: \$145,024.06
Casa de 120 m² → Precio: \$175,430.47
Casa de 150 m² → Precio: \$221,040.09
Casa de 180 m² → Precio: \$266,649.71
=====

5.8 Casos de uso comunes para modelos PKL

1. **Sistemas de producción:** Desplegar modelos en servidores web o APIs
2. **Aplicaciones web:** Usar modelos en Flask, FastAPI o Django
3. **Scripts automatizados:** Ejecutar predicciones en lotes (batch)
4. **Análisis ad-hoc:** Cargar modelos pre-entrenados para análisis rápidos
5. **Compartir modelos:** Enviar modelos entrenados a colegas o clientes
6. **Versionado:** Mantener diferentes versiones de modelos (v1.pkl, v2.pkl, etc.)

5.9 Mejores prácticas

1. **Siempre guarda el escalador:** Los datos nuevos deben escalarse igual que los de entrenamiento
2. **Documenta los metadatos:** Guarda información sobre el modelo (fecha, métricas, versión)
3. **Usa Joblib para scikit-learn:** Es más rápido que Pickle para arrays grandes
4. **Valida antes de desplegar:** Prueba el modelo cargado con datos conocidos
5. **Versiona tus modelos:** Usa nombres descriptivos (modelo_v1.pkl, modelo_20241114.pkl)
6. **Maneja errores:** Incluye try-except al cargar modelos
7. **Verifica compatibilidad:** Asegúrate de usar las mismas versiones de librerías

In [249...]

```

# =====
# EJEMPLO: VERIFICAR MODELO CARGADO
# =====

def verificar_modelo_cargado(ruta_modelo='modelo_completo_casas.pkl', datos_prue
    """
        Verifica que un modelo cargado funcione correctamente
        comparando predicciones con datos de prueba conocidos.
    """

try:

```

```

# Cargar modelo
with open(ruta_modelo, 'rb') as f:
    modelo_completo = pickle.load(f)

    modelo = modelo_completo['modelo']
    scaler = modelo_completo['scaler']

    print("=" * 70)
    print("VERIFICACIÓN DEL MODELO CARGADO")
    print("=" * 70)
    print(f"✓ Modelo cargado exitosamente desde: {ruta_modelo}")
    print(f" - Tipo de modelo: {type(modelo).__name__}")
    print(f" - R² Score guardado: {modelo_completo['r2_score']:.4f}")
    print(f" - RMSE guardado: {modelo_completo['rmse']:.2f}")

# Probar con datos de prueba si están disponibles
if datos_prueba is not None:
    X_test_scaled = scaler.transform(datos_prueba[['Tamaño_m2']])
    predicciones = modelo.predict(X_test_scaled)
    print(f"\n✓ Predicciones generadas correctamente: {len(predicciones)}")
    print(f" Rango de predicciones: ${predicciones.min():,.2f} - ${predicciones.max():,.2f}\n")

    print("\n✓ Modelo verificado y listo para usar")
    print("=" * 70)
    return True

except Exception as e:
    print(f"\nX Error al verificar modelo: {str(e)}")
    print("=" * 70)
    return False

# Verificar modelo cargado
verificar_modelo_cargado('modelo_completo_casas.pkl', X_test)

```

```

=====
VERIFICACIÓN DEL MODELO CARGADO
=====
✓ Modelo cargado exitosamente desde: modelo_completo_casas.pkl
    - Tipo de modelo: LinearRegression
    - R² Score guardado: 0.8852
    - RMSE guardado: 14868.47

✓ Predicciones generadas correctamente: 6 predicciones
    Rango de predicciones: $99,414.44 - $224,080.73

✓ Modelo verificado y listo para usar
=====
```

Out[249...]: True

5.10 Resumen final: Pickle vs Joblib

Característica	Pickle	Joblib
Uso	Estándar de Python	Específico para NumPy/scikit-learn
Velocidad	Más lento	Más rápido para arrays grandes
Tamaño archivo	Más grande	Más pequeño para arrays NumPy

Característica	Pickle	Joblib
Compatibilidad	Universal en Python	Requiere joblib instalado
Recomendado para	Objetos generales	Modelos de scikit-learn

Conclusión: Para modelos de scikit-learn, **Joblib es la mejor opción**. Para otros objetos de Python, usa Pickle.

6. Principales comandos de PyTorch

PyTorch es una librería de Python para redes neuronales y deep learning. Es un framework de machine learning basado en tensores y redes neuronales, desarrollado por Facebook AI Research. Es especialmente útil para deep learning y modelos avanzados de machine learning.

Instalación

```
pip install torch torchvision
```

Importaciones básicas

- `import torch` - Importar PyTorch principal
- `import torch.nn as nn` - Módulo de redes neuronales
- `import torch.optim as optim` - Optimizadores (SGD, Adam, etc.)
- `from torch.utils.data import Dataset, DataLoader` - Para manejo de datos

Creación y manipulación de tensores

- `torch.tensor()` - Crear un tensor desde datos
- `torch.zeros()` - Crear tensor de ceros
- `torch.ones()` - Crear tensor de unos
- `torch.randn()` - Crear tensor con valores aleatorios (distribución normal)
- `torch.arange()` - Crear tensor con rango de valores
- `tensor.shape` - Obtener dimensiones del tensor
- `tensor.dtype` - Obtener tipo de datos del tensor
- `tensor.requires_grad` - Habilitar cálculo de gradientes

Operaciones con tensores

- `tensor1 + tensor2` - Suma elemento a elemento
- `tensor1 * tensor2` - Multiplicación elemento a elemento
- `torch.matmul()` - Multiplicación de matrices
- `tensor.sum()` - Suma de todos los elementos
- `tensor.mean()` - Media de todos los elementos
- `tensor.item()` - Obtener valor escalar de tensor de 1 elemento
- `tensor.numpy()` - Convertir tensor a array de NumPy

- `torch.from_numpy()` - Convertir array de NumPy a tensor

Dispositivos (CPU/GPU)

- `torch.device('cpu')` - Dispositivo CPU
- `torch.device('cuda')` - Dispositivo GPU (si está disponible)
- `torch.cuda.is_available()` - Verificar si GPU está disponible
- `tensor.to(device)` - Mover tensor a dispositivo (CPU o GPU)
- `model.to(device)` - Mover modelo a dispositivo

Redes neuronales (nn.Module)

- `nn.Linear(in_features, out_features)` - Capa completamente conectada
- `nn.ReLU()` - Función de activación ReLU
- `nn.Sigmoid()` - Función de activación Sigmoid
- `nn.Tanh()` - Función de activación Tanh
- `nn.Dropout(p)` - Capa de dropout para regularización
- `nn.Sequential()` - Contenedor para apilar capas secuencialmente

Pérdidas (Loss Functions)

- `nn.MSELoss()` - Error cuadrático medio (para regresión)
- `nn.CrossEntropyLoss()` - Pérdida de entropía cruzada (para clasificación)
- `nn.L1Loss()` - Error absoluto medio (para regresión)
- `nn.BCELoss()` - Pérdida de entropía cruzada binaria

Optimizadores

- `optim.SGD(model.parameters(), lr=0.01)` - Descenso de gradiente estocástico
- `optim.Adam(model.parameters(), lr=0.001)` - Optimizador Adam (recomendado)
- `optim.AdamW()` - Adam con decaimiento de peso
- `optim.RMSprop()` - RMSprop optimizador
- `optimizer.zero_grad()` - Limpiear gradientes acumulados
- `optimizer.step()` - Actualizar parámetros del modelo

Entrenamiento básico

- `loss.backward()` - Calcular gradientes (backpropagation)
- `model.train()` - Modo entrenamiento (habilita dropout, etc.)
- `model.eval()` - Modo evaluación (deshabilita dropout, etc.)
- `torch.no_grad()` - Deshabilitar cálculo de gradientes (para evaluación)

Guardar y cargar modelos

- `torch.save(model.state_dict(), 'modelo.pth')` - Guardar pesos del modelo
- `torch.load('modelo.pth')` - Cargar pesos del modelo
- `model.load_state_dict()` - Cargar pesos en el modelo
- `torch.save(model, 'modelo_completo.pth')` - Guardar modelo completo

Dataset y DataLoader

- `Dataset` - Clase base para datasets personalizados
- `DataLoader(dataset, batch_size, shuffle)` - Cargador de datos por lotes
- `__len__()` - Método para obtener tamaño del dataset
- `__getitem__()` - Método para obtener un elemento del dataset

6.1 ¿Qué es PyTorch?

PyTorch es un framework de machine learning basado en tensores (arrays multidimensionales) que permite construir y entrenar redes neuronales de manera eficiente.

Características principales:

1. **Tensores**: Estructuras de datos similares a arrays de NumPy pero optimizadas para cálculo numérico y deep learning
2. **Diferenciación automática**: Calcula gradientes automáticamente (útil para backpropagation)
3. **Redes neuronales**: Construcción modular de redes neuronales profundas
4. **GPU support**: Acelera cálculos usando tarjetas gráficas (CUDA)
5. **Pythonic**: API intuitiva y fácil de usar, similar a NumPy

Conceptos clave:

- **Tensor**: Array multidimensional (similar a NumPy array pero con capacidades de GPU y diferenciación automática)
- **Autograd**: Sistema de diferenciación automática que calcula gradientes
- **nn.Module**: Clase base para definir redes neuronales
- **Optimizer**: Algoritmo que actualiza los pesos de la red durante el entrenamiento
- **Loss Function**: Función que mide qué tan lejos están las predicciones de los valores reales

Cuándo usar PyTorch:

- **Redes neuronales profundas**: Para modelos complejos con muchas capas
- **Deep learning avanzado**: CNN, RNN, Transformers, etc.
- **Investigación**: Flexibilidad para experimentar con arquitecturas nuevas
- **GPU**: Cuando necesitas aceleración de GPU para entrenar modelos grandes

Ejemplo típico de uso:

```

import torch
import torch.nn as nn

# Definir una red neuronal simple
class RedSimple(nn.Module):
    def __init__(self):
        super().__init__()
        self.capa1 = nn.Linear(10, 64) # 10 entradas, 64 salidas
        self.capa2 = nn.Linear(64, 1) # 64 entradas, 1 salida

    def forward(self, x):
        x = torch.relu(self.capa1(x))
        x = self.capa2(x)
        return x

```

Crear modelo y entrenarlo
 modelo = RedSimple()

En el proyecto, PyTorch se puede usar para crear modelos de regresión más avanzados basados en redes neuronales que pueden capturar relaciones no lineales complejas en los datos.

In [250...]

```

# =====
# EJEMPLO SIMPLE CON PYTORCH
# =====

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import pandas as pd

print("=" * 70)
print("EJEMPLO: REGRESIÓN CON RED NEURONAL EN PYTORCH")
print("=" * 70)

# 1. Crear datos de ejemplo (mismo ejemplo que scikit-Learn para comparación)
np.random.seed(42)
tamaño = np.random.randint(50, 200, 20).astype(float)
precio = (tamaño * 1500 + np.random.randint(-20000, 20000, 20)).astype(float)

print("\n1. Datos de ejemplo creados:")
df_casas = pd.DataFrame({
    'Tamaño_m2': tamaño,
    'Precio': precio
})
print(df_casas.head())
print("\n" + "-" * 50)

```

EJEMPLO: REGRESIÓN CON RED NEURONAL EN PYTORCH

1. Datos de ejemplo creados:

	Tamaño_m2	Precio
0	152.0	213311.0
1	142.0	230819.0
2	64.0	115188.0
3	156.0	231568.0
4	121.0	181269.0

In [251...]

```
# 2. Convertir datos de NumPy a tensores de PyTorch
X = torch.tensor(tamaño.reshape(-1, 1), dtype=torch.float32) # Convertir a tensor X
y = torch.tensor(precio.reshape(-1, 1), dtype=torch.float32) # Convertir a tensor y

print("2. Datos convertidos a tensores de PyTorch:")
print(f"  X shape: {X.shape}") # (20, 1) - 20 muestras, 1 característica
print(f"  y shape: {y.shape}") # (20, 1) - 20 muestras, 1 salida
print(f"  Tipo de X: {X.dtype}")
print("\n" + "-" * 50)
```

2. Datos convertidos a tensores de PyTorch:

```
X shape: torch.Size([20, 1])
y shape: torch.Size([20, 1])
Tipo de X: torch.float32
```

In [252...]

```
# 3. Definir una red neuronal simple para regresión
class RedRegresion(nn.Module):
    """
    Red neuronal simple para regresión:
    - Una capa de entrada (1 característica)
    - Una capa oculta (64 neuronas)
    - Una capa de salida (1 valor)
    """

    def __init__(self):
        super(RedRegresion, self).__init__()
        self.capa_oculta = nn.Linear(1, 64) # 1 entrada -> 64 salidas
        self.activacion = nn.ReLU() # Función de activación
        self.capa_salida = nn.Linear(64, 1) # 64 entradas -> 1 salida

    def forward(self, x):
        # Pasar datos a través de la red
        x = self.capa_oculta(x) # Capa oculta
        x = self.activacion(x) # Activación ReLU
        x = self.capa_salida(x) # Capa de salida
        return x

    # Crear instancia del modelo
    modelo = RedRegresion()
    print("3. Red neuronal creada:")
    print(f"  Modelo: {modelo}")
    print("\n" + "-" * 50)
```

3. Red neuronal creada:

```
Modelo: RedRegresion(
    (capa_oculta): Linear(in_features=1, out_features=64, bias=True)
    (activacion): ReLU()
    (capa_salida): Linear(in_features=64, out_features=1, bias=True)
)
```

In [253...]

```
# 4. Definir función de pérdida y optimizador
criterio = nn.MSELoss() # Error cuadrático medio (para regresión)
optimizador = optim.Adam(modelo.parameters(), lr=0.001) # Optimizador Adam

print("4. Configuración de entrenamiento:")
print(f"  Función de pérdida: {criterio}")
print(f"  Optimizador: {optimizador}")
print("\n" + "-" * 50)
```

4. Configuración de entrenamiento:

```
Función de pérdida: MSELoss()
Optimizador: Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 0.001
    maximize: False
    weight_decay: 0
)
```

In [254...]

```
# 5. Entrenar el modelo
num_epochs = 1000 # Número de iteraciones de entrenamiento
modelo.train() # Poner modelo en modo entrenamiento

print("5. Entrenando modelo...")
for epoch in range(num_epochs):
    # Límpiar gradientes anteriores
    optimizador.zero_grad()

    # Forward pass: calcular predicciones
    predicciones = modelo(X)

    # Calcular pérdida
    perdida = criterio(predicciones, y)

    # Backward pass: calcular gradientes
    perdida.backward()

    # Actualizar pesos
    optimizador.step()

    # Mostrar progreso cada 200 épocas
    if (epoch + 1) % 200 == 0:
```

```

        print(f"    Época {epoch + 1}/{num_epochs}, Pérdida: {perdida.item():.2f}")

print("\n    ✓ Entrenamiento completado")
print("\n" + "-" * 50)

```

5. Entrenando modelo...

Época 200/1000, Pérdida: 43270742016.00
 Época 400/1000, Pérdida: 42693136384.00
 Época 600/1000, Pérdida: 41803726848.00
 Época 800/1000, Pérdida: 40636276736.00
 Época 1000/1000, Pérdida: 39226368000.00

✓ Entrenamiento completado

In [255...]

```

# 6. Hacer predicciones
modelo.eval() # Poner modelo en modo evaluación

# Hacer predicciones (sin calcular gradientes)
with torch.no_grad():
    predicciones = modelo(X)

# Convertir a NumPy para visualización
predicciones_np = predicciones.numpy()
precio_real_np = y.numpy()

print("6. Predicciones del modelo:")
resultados = pd.DataFrame({
    'Tamaño_m2': tamaño,
    'Precio_Real': precio_real_np.flatten(),
    'Precio_Predicho': predicciones_np.flatten()
})
print(resultados.head(10))
print("\n" + "-" * 50)

```

6. Predicciones del modelo:

	Tamaño_m2	Precio_Real	Precio_Predicho
0	152.0	213311.0	11626.219727
1	142.0	230819.0	10864.930664
2	64.0	115188.0	4926.882324
3	156.0	231568.0	11930.734375
4	121.0	181269.0	9266.225586
5	70.0	113693.0	5383.654785
6	152.0	214396.0	11626.219727
7	171.0	263980.0	13072.666992
8	124.0	191658.0	9494.612305
9	137.0	204442.0	10484.286133

In [256...]

```

# 7. Evaluar el modelo con métricas
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

r2 = r2_score(precio_real_np, predicciones_np)
mse = mean_squared_error(precio_real_np, predicciones_np)
rmse = np.sqrt(mse)
mae = mean_absolute_error(precio_real_np, predicciones_np)

print("7. Métricas de evaluación del modelo:")
print(f"    R² Score: {r2:.4f}")

```

```
print(f"    MSE: {mse:.2f}")
print(f"    RMSE: {rmse:.2f}")
print(f"    MAE: {mae:.2f}")
print("\n" + "-" * 50)
```

7. Métricas de evaluación del modelo:

R² Score: -9.4181
MSE: 39218782208.00
RMSE: 198037.33
MAE: 189235.88

In [257...]

```
# 8. Hacer predicción para una nueva casa
nueva_casa = torch.tensor([[100.0]], dtype=torch.float32) # Casa de 100 m2

modelo.eval()
with torch.no_grad():
    precio_predicho = modelo(nueva_casa)

print(f"8. Predicción para casa nueva:")
print(f"    Tamaño: {nueva_casa.item():.0f} m2")
print(f"    Precio predicho: ${precio_predicho.item():,.2f}")
print("\n" + "=" * 70)
```

8. Predicción para casa nueva:

Tamaño: 100 m²
Precio predicho: \$7,667.52

6.3 Comparación: PyTorch vs scikit-learn

Característica	scikit-learn	PyTorch
Uso principal	Machine Learning tradicional	Deep Learning
Facilidad	Muy fácil de usar	Requiere más código
Modelos	Algoritmos clásicos (RF, GBM, etc.)	Redes neuronales personalizadas
GPU	No	Sí (muy rápido)
Gradientes	No necesita	Calcula automáticamente
Casos de uso	Datos tabulares, modelos simples	Datos complejos, imágenes, texto
Recomendado para	Proyectos rápidos, prototipado	Modelos avanzados, investigación

Conclusión: Para la mayoría de problemas de regresión con datos tabulares, **scikit-learn es más simple y suficiente**. Usa **PyTorch** cuando necesites redes neuronales profundas, GPU, o modelos muy personalizados.