PYTHON BÁSICO

1. Ejercicio de codificación : Operaciones Matemáticas
Operaciones Matemáticas Básicas

Objetivo:

Escribe una función en Python llamada `operaciones_matematicas` que reciba dos números enteros `a` y `b`, y realice las siguientes operaciones matemáticas:

1. **Suma**: `a + b`
2. **Resta**: `a - b`
3. **Multiplicación**: `a * b`
4. **División**: `a / b`, si `b` es diferente de 0; de lo contrario, devolver el mensaje `"División por cero no permitida"`.
5. **Resto de la división**: `a % b`, si `b` es diferente de 0; de lo contrario, devolver el mensaje `"División por cero no permitida"`.

La función debe devolver estos cinco resultados en una tupla.

Ejemplo de entrada y salida:

```
a = 10
b = 3
resultado = operaciones_matematicas(a, b)
print("Suma:", resultado[0])          # 13
print("Resta:", resultado[1])         # 7
print("Multiplicación:", resultado[2]) # 30
print("División:", resultado[3])      # 3.3333...
print("Residuo:", resultado[4])       # 1
```

Requisitos:

- La función debe manejar correctamente la división por cero.
- Debe devolver los resultados como una tupla `(suma, resta, multiplicación, división, resto)`.
- Prueba la función con diferentes valores de entrada para verificar su correcto funcionamiento.

2. Calcular la suma y promedio de una lista de números
□ Objetivo

Crear una función en Python que reciba una lista de números y devuelva un diccionario con dos resultados:

- La **suma total** de todos los elementos.
- El **promedio** de esos elementos.

□ Instrucciones

1. Define una función llamada `calcular_suma_y_promedio` que reciba una lista de números como parámetro.
2. Dentro de la función:
   - Crea una variable para guardar la **suma** de los elementos, comenzando en 0.
   - Crea una variable para contar cuántos elementos hay en la lista.
3. Recorre cada número de la lista utilizando un bucle `for`:
   - Suma el número actual a la variable `suma`.
   - Aumenta el contador en 1 por cada número recorrido.
4. Una vez terminado el bucle:
   - Calcula el **promedio** dividiendo la suma total entre el número de elementos (contador).
   - **Importante:** si la lista está vacía, el promedio debe ser 0 para evitar errores.
5. Devuelve un **diccionario** con dos claves:
   - `"suma"`: el total de la suma.
   - `"promedio"`: el valor del promedio.
6. Fuera de la función, crea una lista de prueba llamada `numeros` con algunos valores, por ejemplo `[1, 2, 3, 4, 5]`.
7. Llama a la función con esa lista y muestra por pantalla los resultados accediendo a las claves del diccionario.

□ Ejemplo de uso

```
# Pruebas
numeros = [1, 2, 3, 4, 5]
resultado = calcular_suma_y_promedio(numeros)
print("Suma:", resultado["suma"])
print("Promedio:", resultado["promedio"])
```

```
Suma: 15
```

```
Promedio: 3.0
```

3. Frecuencia de elementos en una lista
□ Objetivo

Crear una función que reciba una lista de elementos y devuelva un diccionario con la frecuencia de aparición de cada elemento en la lista.

□ Instrucciones

1. **Definir la función**:
   - Llama a la función `contar_frecuencia` que tomará como parámetro una lista de elementos.
2. **Inicializar un diccionario**:
   - Dentro de la función, crea un diccionario vacío llamado `frecuencia` para almacenar la cantidad de veces que cada elemento aparece en la lista.
3. **Usar un bucle `while`**:
   - Utiliza un bucle `while` para recorrer la lista de elementos. El bucle debe continuar mientras no se haya recorrido toda la lista.
4. **Comprobar si el elemento ya está en el diccionario**:
   - Para cada elemento de la lista:
     - Si el elemento ya está en el diccionario, **incrementa su frecuencia** en 1.
     - Si el elemento no está en el diccionario, **agrega el elemento** con una frecuencia inicial de 1.
5. **Incrementar el índice**:
   - Al final de cada iteración del bucle, incrementa el índice `i` para pasar al siguiente elemento de la lista.
6. **Devolver el diccionario**:
   - Al finalizar el recorrido de la lista, devuelve el diccionario `frecuencia` que contiene la cantidad de veces que cada elemento aparece en la lista.
7. **Prueba la función**:
   - Fuera de la función, crea una lista con algunos elementos repetidos, por ejemplo `[1, 2, 2, 3, 1, 2, 4, 5, 4]`.

□ Ejemplo

```
# Ejemplo de uso
elementos = [1, 2, 2, 3, 1, 2, 4, 5, 4]
resultado = contar_frecuencia(elementos)
print(resultado)
```

Salida esperada

```
{1: 2, 2: 3, 3: 1, 4: 2, 5: 1}
```

4. Aplicar una Función y Filtrar Elementos en una Lista
🎯 Objetivo

Crear una función que reciba una lista de números y un valor umbral. La función debe aplicar una operación a cada número (como elevarlo al cuadrado) y luego filtrar aquellos resultados que sean mayores que el valor umbral.

📝 Instrucciones

1. **Definir la función**:
   - Llama a la función `aplicar_funcion_y_filtrar`, que tomará como parámetros una lista de números `lista` y un número `valor_umbral`.
2. **Aplicar una función con `map`**:
   - Usa la función `map` para aplicar una operación a cada elemento de la lista. En este caso, la operación será elevar al cuadrado cada número de la lista.
3. **Filtrar los resultados con `filter`**:
   - Después de aplicar la función, usa la función `filter` para filtrar los elementos que sean mayores que el valor umbral (`valor_umbral`).
4. **Devolver los resultados**:
   - La función debe devolver una lista con los elementos que pasen el filtro.
5. **Prueba la función**:
   - Fuera de la función, crea una lista de números, por ejemplo `[1, 2, 3, 4, 5]`, y un valor umbral (por ejemplo, 3).
   - Llama a la función con esta lista y umbral, y muestra el resultado.

Ejemplo de entrada y salida:

```
numeros = [1, 2, 3, 4, 5]
valor_umbral = 3
resultado = aplicar_funcion_y_filtrar(numeros, valor_umbral)
print(resultado)
```

Salida esperada:

`[4, 9, 16, 25]`

NUMPY

5. Generar N números aleatorios enteros entre un valor mínimo y máximo
Desarrolla una función llamada `generar_numeros_enteros_aleatorios` que tome como entrada el número de elementos `N`, un valor mínimo `minimo` y un valor máximo `maximo`, y utilice NumPy para generar una lista de `N` números enteros aleatorios en el rango `[minimo, maximo]`.

```
def generar_numeros_enteros_aleatorios(N, minimo, maximo):
```

```
# Ejemplo de uso
N = 5
minimo = 1
maximo = 10
resultado = generar_numeros_enteros_aleatorios(N, minimo, maximo)
print(resultado)
```

Resultado:

```
[4, 10, 1, 8, 2]
```

6. Generar una Secuencia de Números
Desarrolla una función llamada `generar_secuencia_numerica` que tome como entrada un valor mínimo `minimo`, un valor máximo `maximo` y un paso `paso`, y utilice NumPy para generar una secuencia de números en el rango `[minimo, maximo)` con el paso especificado.

```
def generar_secuencia_numerica(minimo, maximo, paso):
```

```
# Ejemplo de uso
minimo = 0
maximo = 10
paso = 2
resultado = generar_secuencia_numerica(minimo, maximo, paso)
print(resultado)
```

Resultado

```
[0 2 4 6 8]
```

Tipo de datos del resultado

```
type(resultado)
```

```
numpy.ndarray
```

7. Finanzas personales
Crear una función en Python que realice operaciones con arrays utilizando **NumPy** para analizar las finanzas personales de un estudiante. La función debe recibir dos arrays: uno con los ingresos mensuales de un estudiante durante un año y otro con sus gastos mensuales. La función debe devolver el balance mensual (ingresos - gastos), la suma total de los ingresos, la suma total de los gastos, y el saldo total (ingresos totales - gastos totales) durante todo el año.

Instrucciones:

1. **Crear la función `analizar_finanzas`:**
    - La función debe recibir dos arrays de **12 elementos**: uno con los ingresos mensuales (`ingresos`) y otro con los gastos mensuales (`gastos`).
    - La función debe realizar las siguientes operaciones:
        - **Balance mensual**: Resta los ingresos menos los gastos para cada mes.
        - **Total de ingresos**: Suma todos los ingresos del año.
        - **Total de gastos**: Suma todos los gastos del año.
        - **Saldo final**: Calcula el saldo final del año (total ingresos - total gastos).
    - La salida debe ser un array con estos 4 resultados:
    1. `resultado = [balance_mensual, total_ingresos, total_gastos, saldo_final]`
2. **Requisitos**:
    - Debes usar la librería **NumPy** para hacer las operaciones de forma eficiente.
    - Los datos de los ingresos y los gastos deben ser positivos (no considerar valores negativos).

Ejemplo de datos de entrada y salida esperada:

1.
- **Ingresos mensuales**: `[1500, 1600, 1700, 1650, 1800, 1900, 2000, 2100, 2200, 2300, 2400, 2500]`
- **Gastos mensuales**: `[1000, 1100, 1200, 1150, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000]`

2. **Salida esperada:** La salida de la función debe ser un array con los resultados de las operaciones, de la siguiente forma:

```
1. resultado = [
2.   [500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500, 500],
3.   23650,
4.   17650,
5.   6000]
```

PANDAS

8. Calcular promedio
Estás trabajando con una tabla de datos numéricos almacenada en un `DataFrame` de pandas. El objetivo es calcular el **promedio de cada columna** de manera automatizada, sin usar bucles explícitos.

Sigue estos pasos para construir la solución:

1. **Crea una función llamada** `calcular_promedio(dataframe)` que reciba un `DataFrame` como parámetro.
2. Dentro de la función, **usa un método de pandas que calcule el promedio de cada columna automáticamente**. Evita usar bucles `for` o `while`.
3. Retorna los valores promedio como resultado de la función.
4. Prueba tu función con un `DataFrame` de ejemplo que contenga varias columnas con números enteros.
5. Imprime el resultado y asegúrate de que sea una **Serie de pandas** con el promedio de cada columna.

```
# Ejemplo de uso
data = {
    'A': [1, 2, 3, 4],
    'B': [5, 6, 7, 8],
    'C': [9, 10, 11, 12]
}

df = pd.DataFrame(data)
resultado = calcular_promedio(df)
print(resultado)
```

Resultado

```
A     2.5
B     6.5
C    10.5
dtype: float64
```

9. Seleccionar datos
Objetivo:
El objetivo de este ejercicio es familiarizarte con la manipulación de datos en pandas, utilizando filtros y criterios personalizados para seleccionar subconjuntos de datos en un DataFrame.

Contexto:
Imagina que estás trabajando con un conjunto de datos de estudiantes, donde cada registro contiene el nombre, la edad y las calificaciones de cada uno. Tienes la tarea de implementar una función que permita seleccionar ciertos registros de acuerdo con un criterio dado (por ejemplo, edad mayor a 18 años).

Tarea:

1. **Crea una función** llamada `seleccionar_datos(dataframe, criterio)` que tome dos argumentos:
   - `dataframe`: un DataFrame de pandas que contiene varios registros.
   - `criterio`: una cadena que representa una expresión de consulta que pandas puede entender (como `edad > 18` o `calificaciones >= 90`).

   La función debe usar este criterio para filtrar los datos y devolver solo los registros que cumplan con el criterio.

Requisitos:

- Utiliza el método `query()` de pandas para aplicar el filtro de manera eficiente.
- Asegúrate de que la función sea flexible y permita cambiar el criterio de selección.

```
# Ejemplo de uso
data = {
    'nombre': ['Alice', 'Bob', 'Charlie', 'David'],
    'edad': [20, 22, 18, 25],
    'calificaciones': [90, 88, 75, 95]
}

df = pd.DataFrame(data)
criterio = 'edad > 18'
resultado = seleccionar_datos(df, criterio)
print(resultado)
```

```
   nombre  edad  calificaciones
0  Alice    20              90
1    Bob    22              88
3  David    25              95
```

10. Limpieza inteligente de datos
Imagina que estás trabajando con un conjunto de datos reales sobre estudiantes, pero algunas calificaciones están incompletas (aparecen como valores nulos). Tu tarea será completar esos valores vacíos de forma inteligente, utilizando la **media (promedio)** de la columna como reemplazo.

 Tu objetivo:

Implementa una función llamada `rellenar_con_media(dataframe, columna)` que:

1. Reciba como argumentos un `DataFrame` de pandas y el **nombre de una columna numérica** (por ejemplo, `'calificaciones'`).
2. Calcule la media de los valores **no nulos** de esa columna.
3. Reemplace todos los valores nulos (`NaN`) en esa columna con la **media calculada**.
4. Devuelva el `DataFrame` modificado.

 Ejemplo de uso:

```
data = {
    'nombre': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'edad': [20, None, 18, 25, None],
    'calificaciones': [90, 88, None, None, 95]
}

df = pd.DataFrame(data)
columna = 'calificaciones'

resultado = rellenar_con_media(df, columna)
print(resultado)
```

 Salida esperada

```
    nombre  edad  calificaciones
0    Alice  20.0            90.0
1      Bob   NaN            88.0
2  Charlie  18.0            91.0
3    David  25.0            91.0
4      Eve   NaN            95.0
```

La columna `"calificaciones"` debe tener sus valores `NaN` reemplazados por la media de las calificaciones conocidas.

MATPLOTLIB

11. Horas de estudio y calificaciones
Crear una función en Python utilizando `matplotlib` que grafique la relación entre las horas de estudio y las calificaciones obtenidas por los estudiantes, con el objetivo de visualizar la tendencia de cómo las calificaciones aumentan a medida que aumentan las horas de estudio.

Instrucciones:

1. **Crear la función `graficar_linea`:**
    - Define una función llamada `graficar_linea` que reciba dos listas:
        - `x`: una lista que contenga los valores de las horas de estudio.
        - `y`: una lista que contenga las calificaciones obtenidas.
    - La función debe utilizar la librería `matplotlib.pyplot` para graficar una línea que conecte los puntos `(x, y)` en un gráfico de dispersión.
    - La gráfica debe incluir:
        - Un título: `"Relación entre horas de estudio y calificaciones"`.
        - Etiquetas en los ejes:
            - El eje X debe estar etiquetado como `"Horas de Estudio"`.
            - El eje Y debe estar etiquetado como `"Calificación"`.
2. **Datos de entrada:** A continuación se proporcionan los datos que debes usar para graficar la relación entre las horas de estudio y las calificaciones.
    - **Horas de estudio (x)**: `[1, 2, 3, 4, 5, 6, 7, 8]`
    - **Calificaciones (y)**: `[55, 60, 65, 70, 75, 80, 85, 90]`
3. **Requisitos:**
    - Usa la librería `matplotlib` para crear la gráfica.
    - La gráfica debe ser una línea que conecte los puntos `(x, y)`.
    - Al ejecutar la función, debe aparecer una ventana con la gráfica generada.
    - Asegúrate de que los ejes X e Y estén correctamente etiquetados y que el gráfico tenga el título especificado.
4. **Prueba:**
    - Llama a la función `graficar_linea` con los datos proporcionados y asegúrate de que la gráfica se muestre correctamente.

12. Temperaturas semanales
Crear una función que reciba una lista de temperaturas diarias y los días de la semana correspondientes, y genere un gráfico de líneas utilizando **Matplotlib**. El gráfico debe visualizar las temperaturas a lo largo de la semana.

Requisitos:

1. **Función a crear:**
   La función debe recibir dos listas:
   - `dias`: Una lista de los días de la semana (por ejemplo, ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']).
   - `temperaturas`: Una lista de las temperaturas correspondientes a cada día de la semana (por ejemplo, [22, 24, 23, 25, 26, 28, 27]).
2. **Gráfico:**
   - Utilizar **Matplotlib** para graficar las temperaturas.
   - El gráfico debe tener los días de la semana en el eje X y las temperaturas en el eje Y.
   - Configurar el título del gráfico como **"Temperaturas Semanales"**.
   - El gráfico debe tener etiquetas en los ejes (por ejemplo, "Días" en el eje X y "Temperatura (°C)" en el eje Y).
   - Personalizar el gráfico cambiando el color de la línea a **azul** y utilizando un estilo de línea **dashed** (discontinua).
3. **Opcional:**
   - Añadir una leyenda al gráfico que indique "Temperatura".
   - Modificar el tamaño del gráfico (por ejemplo, hacerlo más grande).

Ejemplo de Entrada y Salida:

- **Entrada:**

```
dias = ['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
temperaturas = [22, 24, 23, 25, 26, 28, 27]
```

- **Salida:**
  Un gráfico que muestra las temperaturas a lo largo de la semana, con líneas discontinuas azules y el título "Temperaturas Semanales".

REGRESIÓN LINEAL

13. Regresión Lineal con Datos de Ventas
Supongamos que tienes un conjunto de datos que contiene información sobre las ventas de una tienda y la cantidad de dinero que se gastó en publicidad en diferentes medios (por ejemplo, televisión, radio y periódico).

Tu tarea es desarrollar una función llamada `regresion_ventas` que tome estos datos como entrada y ajuste un modelo de regresión lineal para predecir las futuras ventas en función de la inversión en publicidad.

```
def regresion_ventas(datos):
```

```
# Ejemplo de uso con datos reales
data = {
    'TV': [230.1, 44.5, 17.2, 151.5, 180.8],
    'Radio': [37.8, 39.3, 45.9, 41.3, 10.8],
    'Periodico': [69.2, 45.1, 69.3, 58.5, 58.4],
    'Ventas': [22.1, 10.4, 9.3, 18.5, 12.9]
}
df = pd.DataFrame(data)
modelo_regresion = regresion_ventas(df)

# Estimaciones de ventas para nuevos datos de inversión en publicidad
nuevos_datos = pd.DataFrame({'TV': [200, 60, 30], 'Radio': [40, 20, 10], 'Periodico': [50, 10, 5]})
estimaciones_ventas = modelo_regresion.predict(nuevos_datos)

print("Estimaciones de Ventas:")
print(estimaciones_ventas)
```

Resultado:

```
Estimaciones de Ventas:
[21.54261464  8.48121675  4.16961329]
```

14. Predecir el rendimiento de un jugador
□ Enunciado del ejercicio

Imagina que formas parte del equipo de desarrollo de un videojuego multijugador competitivo. El equipo de analítica necesita predecir cuántas **victorias** podría lograr un nuevo jugador, basándose únicamente en su estilo de juego.

Como científico de datos, tu misión es crear un modelo de **regresión lineal** que pueda predecir las victorias de un jugador en función de su rendimiento medio en partidas anteriores.

Para organizar bien tu solución y facilitar su reutilización en el futuro, deberás construirla utilizando tres clases bien definidas.

□□ Lo que debes hacer

1. Crea la clase `Player`

Esta clase representa a un jugador. Debe contener:

- `name`: nombre del jugador
- `avg_session_time`: duración promedio de sus sesiones de juego (en minutos)
- `avg_actions_per_min`: acciones por minuto que realiza
- `avg_kills_per_session`: número promedio de eliminaciones por sesión
- `victories`: número de victorias (opcional, ya que puede usarse para predicción)

Debe incluir un método `to_features(self)` que devuelva una lista con los valores de entrada para el modelo.

2. Crea la clase `PlayerDataset`

Esta clase representa una colección de jugadores. Debe tener:

- Un constructor que reciba una lista de objetos `Player`
- Un método `get_feature_matrix()` que devuelva una lista de listas con los valores de entrada (`X`)
- Un método `get_target_vector()` que devuelva una lista con los valores objetivo (`y`, las victorias)

3. Crea la clase `VictoryPredictor`

Encargada de entrenar y usar el modelo de regresión. Debe contener:

- Un atributo con el modelo (`LinearRegression`)

- Un método `train(dataset: PlayerDataset)` para entrenar el modelo con los datos del dataset
- Un método `predict(player: Player)` que devuelva el número de victorias predichas para ese jugador

Ejemplo de uso

```
players = [
    Player("Alice", 40, 50, 6, 20),
    Player("Bob", 30, 35, 4, 10),
    Player("Charlie", 50, 60, 7, 25),
    Player("Diana", 20, 25, 2, 5),
    Player("Eve", 60, 70, 8, 30)
]

dataset = PlayerDataset(players)
predictor = VictoryPredictor()
predictor.train(dataset)

test_player = Player("TestPlayer", 45, 55, 5)
predicted = predictor.predict(test_player)
print(f"Victorias predichas para {test_player.name}: {predicted:.2f}")
```

Salida esperada

```
Victorias predichas para TestPlayer: 22.50
```

15. Predecir ingresos de una aplicación
□ Enunciado del ejercicio:

Eres parte de un equipo de analistas de datos en una empresa tecnológica que desarrolla aplicaciones móviles. Te han proporcionado un pequeño conjunto de datos con información sobre diferentes apps que ya están publicadas, y tu tarea es crear un modelo de **regresión lineal** para **predecir los ingresos estimados** de una nueva app.

□ Datos disponibles por app:

- `app_name`: Nombre de la app
- `downloads`: Número de descargas (en miles)
- `rating`: Valoración media de los usuarios (de 1 a 5)
- `size_mb`: Tamaño de la app (en MB)
- `reviews`: Número de valoraciones escritas
- `revenue`: Ingresos generados (en miles de dólares) → **variable a predecir**

□ Tareas que debes realizar:

1. Crea una clase `App` que represente cada app con sus atributos.
2. Crea una clase `RevenuePredictor` que:
    - Reciba una lista de objetos `App`.
    - Extraiga las características relevantes para entrenar un modelo.
    - Entrene un modelo de **regresión lineal** para predecir los ingresos (`revenue`).
    - Permita predecir los ingresos de una nueva app con datos similares.
3. Entrena el modelo con los datos proporcionados (puedes usar una lista de ejemplo en el código).
4. Prueba el modelo prediciendo los ingresos estimados de una nueva app ficticia.

□ Ejemplo de uso

```
# Datos simulados de entrenamiento
training_apps = [
    App("TaskPro", 200, 4.2, 45.0, 1800, 120.0),
    App("MindSpark", 150, 4.5, 60.0, 2100, 135.0),
```

```
    App("WorkFlow", 300, 4.1, 55.0, 2500, 160.0),
    App("ZenTime", 120, 4.8, 40.0, 1700, 140.0),
    App("FocusApp", 180, 4.3, 52.0, 1900, 130.0),
    App("BoostApp", 220, 4.0, 48.0, 2300, 145.0),
]

# Creamos y entrenamos el predictor
predictor = RevenuePredictor()
predictor.fit(training_apps)

# Nueva app para predecir
new_app = App("FocusMaster", 250, 4.5, 50.0, 3000)
predicted_revenue = predictor.predict(new_app)

print(f"Ingresos estimados para {new_app.name}: ${predicted_revenue:.2f}K")
```

Salida esperada

```
Ingresos estimados para FocusMaster: $207.59K
```

16. Predicción del desgaste de vehículos
□ Misión: Predicción del Desgaste de Vehículos Militares

Como analista de datos en una base militar, tu tarea es predecir el **nivel de desgaste** de vehículos en función de las **horas de uso**.

Esta herramienta será clave para evitar fallos operativos y optimizar los mantenimientos preventivos.

Debes implementar un sistema basado en regresión lineal que:

1. Genere registros de entrenamiento con la clase `VehicleDataGenerator`.
2. Represente cada registro con la clase `VehicleRecord`, que almacene las horas de uso y el nivel de desgaste.
3. Entrene un modelo con la clase `VehicleWearRegressor` para aprender la relación entre las horas y el desgaste.
4. Visualice los resultados y permita hacer predicciones con la clase principal `VehicleWearPredictionExample`.

Detalles técnicos:

- Usa `NumPy`, `Pandas`, `Matplotlib` y `LinearRegression` de `scikit-learn`.
- Los datos sintéticos deben simular que el desgaste aumenta con las horas de uso, con algo de ruido.
- El modelo debe permitir predecir el nivel de desgaste de un nuevo vehículo dado sus horas de servicio.
- Visualiza los datos reales y la línea de regresión, y marca en el gráfico la predicción del nuevo vehículo.
- Organiza la solución en **clases**, siguiendo estos nombres:

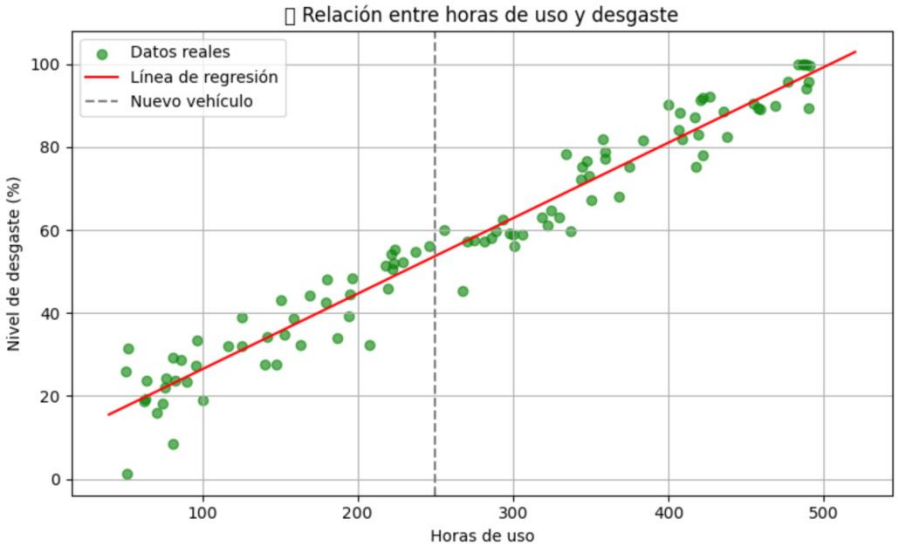
Propósito	Nombre de la clase
Representar un registro de vehículo	`VehicleRecord`
Generar datos sintéticos	`VehicleDataGenerator`
Entrenar y usar el modelo	`VehicleWearRegressor`
Ejecutar el ejemplo completo	`VehicleWearPredictionExample`

Ejemplo de uso

```
example = VehicleWearPredictionExample()
example.run()
```

Salida esperada

```
 Horas de uso estimadas: 250
  Nivel de desgaste estimado: 53.75%
```



17. Predecir consumo energético de satélites
□ Objetivo

Una agencia espacial quiere entender mejor cómo varía el **consumo energético diario** de sus satélites dependiendo de la **duración de su misión**.

Te han pedido que analices esta relación utilizando **datos simulados**, y que desarrolles un modelo que permita:

1. Generar una muestra de satélites con características realistas.
2. Calcular una **nueva métrica**: la **eficiencia energética** del satélite.
3. Aplicar un modelo de **regresión lineal** para predecir el consumo diario a partir de la duración de la misión.
4. Visualizar los resultados para facilitar la toma de decisiones.

□ Actividades a realizar

1. Crear una clase `Satellite`

Esta clase debe representar un satélite individual con los siguientes atributos:

- `duracion_mision` (en días)
- `paneles_sol` (superficie total de paneles solares en m^2)
- `carga_util` (peso de instrumentos y sensores en kg)
- `consumo_diario` (energía consumida por día en kWh)

2. Generar datos sintéticos con `SatelliteDatasetGenerator`

Esta clase debe generar N satélites simulados (por defecto 300), con los siguientes criterios:

- `duracion_mision`: aleatorio entre 100 y 1000 días.
- `paneles_sol`: aleatorio entre 10 y 100 m^2.
- `carga_util`: aleatorio entre 200 y 2000 kg.
- `consumo_diario`: calculado con la fórmula:

consumo_diario=5+(0.01·duracion_mision)+(0.002·carga_util)+ruido_normal

Donde `ruido_normal` es una perturbación aleatoria generada con `np.random.normal(0, 1)` para hacer los datos más realistas.

3. Procesar los datos con `SatelliteDataProcessor`

Convierte la lista de objetos `Satellite` en un `DataFrame` de pandas, y añade una columna nueva:

$$\text{eficiencia\_energia} = \frac{\text{consumo\_diario}}{\text{paneles\_sol}}$$

Esta métrica mide cuánto consume el satélite por cada metro cuadrado de panel solar.

4. Aplicar regresión lineal con `EnergyConsumptionRegressor`

Esta clase debe aplicar el modelo de regresión lineal para predecir el consumo energético a partir de la duración de la misión. Debes:

- Entrenar el modelo con `duracion_mision_dias` como entrada (`X`) y `consumo_diario` como salida (`y`).
- Calcular el R^2 del modelo (`r2_score`) y mostrar los coeficientes:

y_pred=(coef·duracion_mision)+intercepto

5. Visualizar los resultados con `SatellitePlotter`

Muestra un gráfico con:

- Puntos dispersos: consumo diario vs. duración de la misión.
- Color por carga útil (usa un mapa de color `viridis`).
- Línea de regresión en rojo.

6. Ejecutar todo con la clase `SatelliteAnalysisExample`

Una vez que has desarrollado las clases anteriores para generar, procesar, modelar y visualizar los datos, el último paso es **integrar todo** en una clase orquestadora que automatice el análisis completo.

Esta clase debe encargarse de:

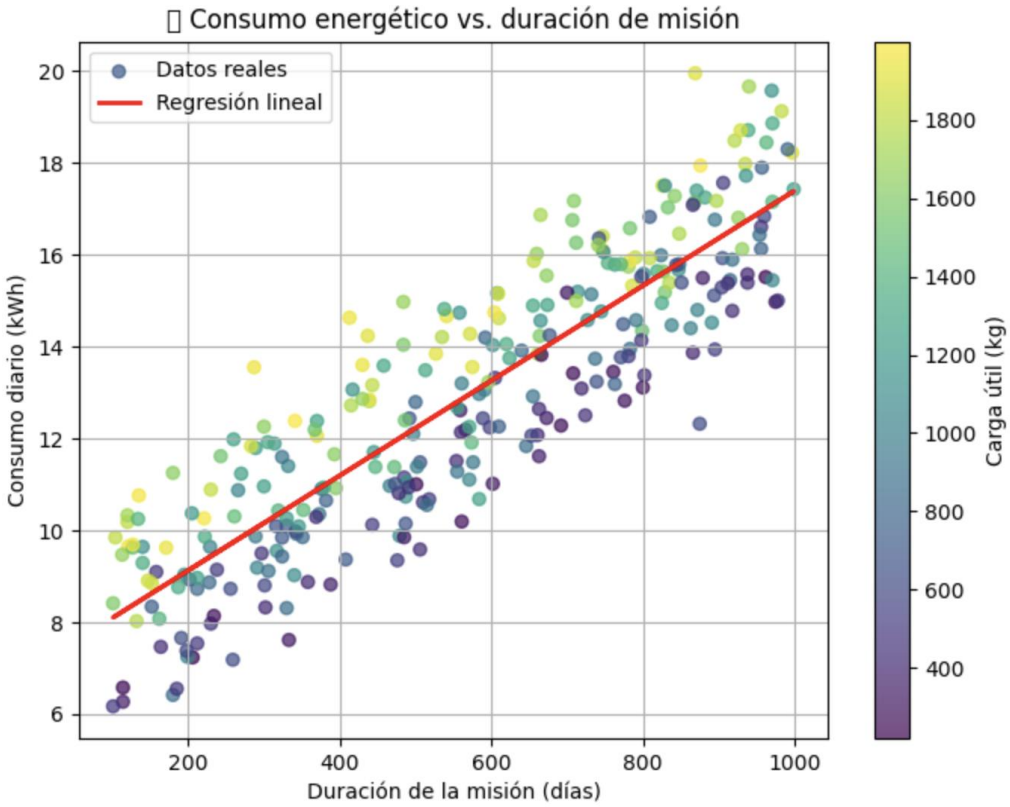
1. **Generar los datos**: usando la clase `SatelliteDatasetGenerator`.
2. **Procesarlos**: con `SatelliteDataProcessor`, que además de convertir la lista a `DataFrame`, añade la columna de eficiencia.
3. **Aplicar regresión**: usando `EnergyConsumptionRegressor` para entrenar el modelo con la duración de la misión como predictor.
4. **Evaluar el modelo**: imprimiendo el coeficiente de regresión, el intercepto y el **R^2**, que indica qué tan bien se ajusta el modelo a los datos.
5. **Visualizar los resultados**: con `SatellitePlotter`, mostrando una gráfica clara con los datos y la predicción lineal

□ Ejemplo de uso

```
example = SatelliteAnalysisExample()
example.run()
```

Salida esperada

```
□ Modelo: y = 0.0103 * x + 7.06
□ R² del modelo: 0.7780
```



K VECINOS MÁS CERCANOS

18. Clasificación Binaria
Supongamos que tienes un conjunto de datos que contiene información sobre pacientes y deseas predecir si un paciente tiene una enfermedad (1) o no (0) en función de algunas características médicas, como la edad y los niveles de colesterol.

Tu tarea es desarrollar una función llamada `regresion_logistica` que tome estos datos como entrada y ajuste un modelo de regresión logística para la clasificación binaria.

```
def regresion_logistica(datos):
```

```
# Ejemplo de uso con datos de pacientes
data = {
    'Edad': [50, 35, 65, 28, 60],
    'Colesterol': [180, 150, 210, 130, 190],
    'Enfermedad': [1, 0, 1, 0, 1]
}

df = pd.DataFrame(data)
modelo_regresion_logistica = regresion_logistica(df)

# Estimaciones de clasificación binaria para nuevos datos
nuevos_datos = pd.DataFrame({'Edad': [45, 55], 'Colesterol': [170, 200]})
estimaciones_clasificacion = modelo_regresion_logistica.predict(nuevos_datos)
print("Estimaciones de Clasificación:")
print(estimaciones_clasificacion)
```

Resultados:

```
Estimaciones de Clasificación:
[1 1]
```

19. Predecir futuro de una app
¿Tendrá éxito tu app?

 Contexto

Eres parte de un equipo de análisis de una startup que lanza apps móviles. Se te ha asignado la tarea de construir un modelo que pueda **predecir si una app será exitosa** o no en función de sus métricas iniciales.

La empresa ha recopilado datos de otras apps anteriores, tanto exitosas como fallidas, y quiere automatizar este análisis con Machine Learning.

 Objetivo

Crea un sistema en Python que permita:

1. Representar los datos de una app.
2. Preparar un conjunto de datos a partir de múltiples apps.
3. Entrenar un modelo de **regresión logística** con `scikit-learn`.
4. Predecir si una app será exitosa.
5. De forma opcional, mostrar la **probabilidad de éxito**.

 Estructura del proyecto

Debes implementar las siguientes clases:

 `App`

Representa una app móvil con las siguientes características:

- `app_name`: nombre de la app.
- `monthly_users`: número de usuarios mensuales.
- `avg_session_length`: duración media de las sesiones (en minutos).
- `retention_rate`: tasa de retención entre 0 y 1.
- `social_shares`: número de veces que se ha compartido en redes sociales.
- `success`: valor opcional (1 = éxito, 0 = fracaso).

Método:

- `to_features(self)`: devuelve una lista de características numéricas.

 `AppDataset`

Representa un conjunto de datos de apps. Debe incluir:

- Lista de objetos `App`.

Métodos:

- `get_feature_matrix(self)`: devuelve una matriz de características.
- `get_target_vector(self)`: devuelve un vector de etiquetas (success).

 `SuccessPredictor`

Encargado de entrenar y usar el modelo de regresión logística.

Métodos:

- `train(dataset)`: entrena el modelo usando un `AppDataset`.
- `predict(app)`: devuelve 1 o 0 para predecir si la app será exitosa.
- `predict_proba(app)`: (opcional) devuelve la probabilidad de éxito como número decimal entre 0 y 1.

 Sugerencia: puedes usar `StandardScaler` para mejorar la precisión del modelo escalando los datos.

 Ejemplo de uso

```
# Datos de entrenamiento
apps = [
    App("FastChat", 10000, 12.5, 0.65, 1500, 1),
    App("FitTrack", 500, 5.0, 0.2, 50, 0),
    App("GameHub", 15000, 25.0, 0.75, 3000, 1),
    App("BudgetBuddy", 800, 6.5, 0.3, 80, 0),
    App("EduFlash", 12000, 18.0, 0.7, 2200, 1),
    App("NoteKeeper", 600, 4.0, 0.15, 30, 0)
]

dataset = AppDataset(apps)
predictor = SuccessPredictor()
predictor.train(dataset)

# Nueva app a evaluar
new_app = App("StudyBoost", 20000, 15.0, 0.5, 700)
predicted_success = predictor.predict(new_app)
prob = predictor.predict_proba(new_app)

print(f"¿Será exitosa la app {new_app.app_name}? {'Sí' if predicted_success else 'No'}")
print(f"Probabilidad estimada de éxito: {prob:.2f}")
```

 Salida esperada

```
¿Será exitosa la app StudyBoost? Sí
Probabilidad estimada de éxito: 0.83
```

20. Predecir resultados en partidas multijugador
 Objetivo

En este ejercicio, aplicarás tus conocimientos de regresión logística para construir un modelo capaz de predecir si un jugador ganó o perdió una partida, a partir de sus estadísticas individuales.

 Descripción del problema

Tienes que construir un modelo predictivo que, a partir de las estadísticas de un jugador en una partida, determine si ganó o no. Para ello, deberás:

- Crear datos sintéticos que representen partidas ficticias de jugadores.
- Entrenar un modelo de regresión logística con esos datos.
- Implementar una función que prediga el resultado (ganar o no) para un nuevo jugador.

□ Paso 1: Definir una clase para representar una partida

Crea una clase `PlayerMatchData` con los siguientes atributos:

- `kills`: número de enemigos eliminados
- `deaths`: número de veces que el jugador ha muerto
- `assists`: asistencias realizadas
- `damage_dealt`: daño total infligido
- `damage_received`: daño total recibido
- `healing_done`: curación realizada
- `objective_time`: tiempo (en segundos) que el jugador estuvo capturando objetivos
- `won`: `1` si el jugador ganó la partida, `0` si perdió

Incluye un método `.to_dict()` que devuelva los datos como un diccionario (sin la variable `won`, opcionalmente).

□ Paso 2: Generar datos sintéticos con NumPy

Crea una función llamada `generate_synthetic_data` que genere un conjunto de datos de entrenamiento simulando partidas de videojuegos. Para ello:

- Utiliza la librería `numpy` para generar los valores numéricos.
- Cada instancia representará el desempeño de un jugador en una partida.
- La función debe devolver una lista de objetos `PlayerMatchData` (ya definida previamente).
- Implementa la siguiente lógica para cada jugador:

Reglas para los datos:

- `kills`: número de enemigos eliminados, generado con una distribución de Poisson con media 5.
    - 1. `kills = np.random.poisson(5)`
- `deaths`: número de veces que el jugador ha muerto, distribución de Poisson con media 3.
- `assists`: asistencias realizadas, distribución de Poisson con media 2.
- `damage_dealt`: daño infligido, calculado como `kills * 300 + ruido aleatorio normal`.
    - 1. `damage_received = deaths * 400 + np.random.normal(0, 100)`
- `damage_received`: daño recibido, como `deaths * 400 + ruido aleatorio normal`.

- `healing_done`: cantidad de curación, valor aleatorio entero entre 0 y 300.
- `objective_time`: tiempo (en segundos) controlando objetivos, valor aleatorio entre 0 y 120.
- `won`: el jugador se considera que ganó la partida si hizo más daño del que recibió **y** tuvo más kills que muertes.

 Tu función debe seguir esta estructura:

```
import numpy as np

def generate_synthetic_data(n=100):
    data = []
    for _ in range(n):
        # Genera cada variable siguiendo las instrucciones dadas
        # Crea un objeto PlayerMatchData con estos valores
        # Añádelo a la lista de datos

    return data
```

Paso 3: Crear y entrenar el modelo

Crea una clase `VictoryPredictor` que entrene un modelo de regresión logística con los datos sintéticos. Esta clase debe tener:

- Un método `train(data)` para entrenar el modelo.
- Un método `predict(player: PlayerMatchData)` que devuelva `1` si predice victoria, `0` si derrota.

Ejemplo de uso

```
# Crear datos de entrenamiento
training_data = generate_synthetic_data(150)

# Entrenar modelo
predictor = VictoryPredictor()
predictor.train(training_data)

# Crear jugador de prueba
test_player = PlayerMatchData(8, 2, 3, 2400, 800, 120, 90, None)

# Predecir si ganará
prediction = predictor.predict(test_player)
print(f"¿El jugador ganará? {'Sí' if prediction == 1 else 'No'}")
```

Salida esperada

```
¿El jugador ganará? Sí
```

21. Predicción meteorológica con gráfico
Los meteorólogos recopilan datos de humedad y presión atmosférica para predecir si lloverá o no.

Tu misión es construir un sistema inteligente que pueda predecir la probabilidad de lluvia utilizando un modelo de regresión logística.

Pasos a seguir

1. **Crea la clase `WeatherRecord`:**
   Representará un registro de condiciones meteorológicas con:
   - Humedad (%)
   - Presión atmosférica (hPa)

- Lluvia (1 si llovió, 0 si no)

2. **Crea la clase `WeatherDataGenerator`:**
   Deberá generar registros sintéticos simulando que:
   - Alta humedad y baja presión aumentan la probabilidad de lluvia.
   - Baja humedad y alta presión indican que no lloverá.
3. **Crea la clase `WeatherRainClassifier`:**
   Que entrene un modelo de regresión logística utilizando los datos generados. Debe incluir:
   - Método `fit(records)` para entrenar.
   - Método `predict(humidity, pressure)` que devuelva 1 o 0.
   - Método `get_model()` para obtener el modelo scikit-learn (útil para visualización).
4. **Crea la clase `WeatherPredictionExample`:**
   Que haga lo siguiente:
   - Genere datos.
   - Entrene el modelo.
   - Haga una predicción para nuevas condiciones climáticas.
   - Muestre un gráfico con matplotlib diferenciando los casos de lluvia y no lluvia.

□ Ejemplo de uso

```
example = WeatherRainPredictionExample()
example.run()
```

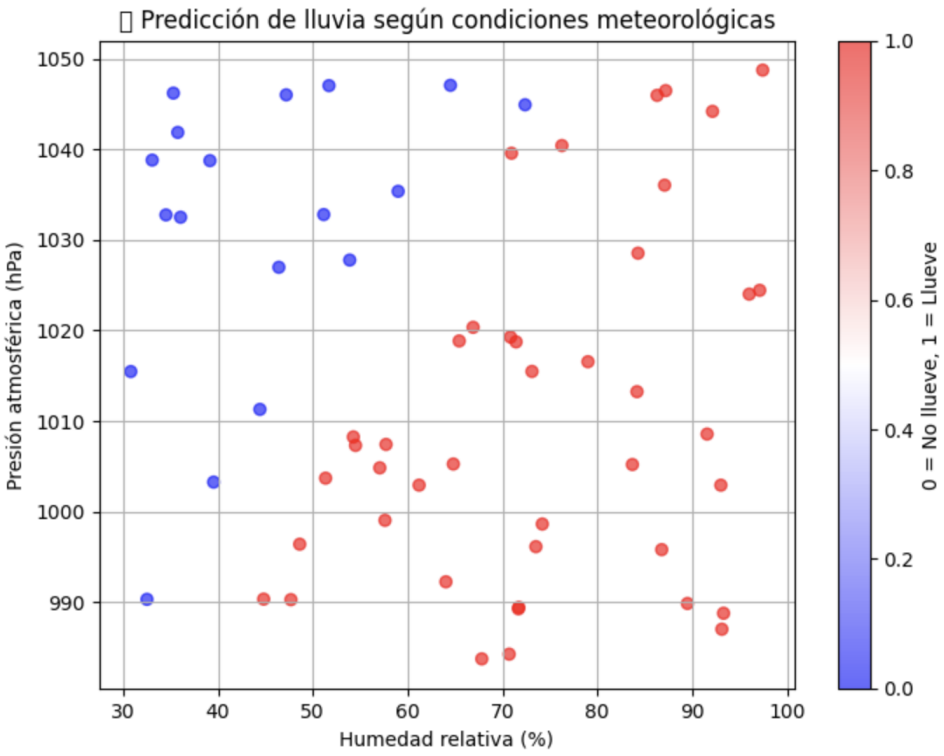
□ Salida esperada

```
[[21  0]
 [ 0 39]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        21
           1       1.00      1.00      1.00        39

    accuracy                           1.00        60
   macro avg       1.00      1.00      1.00        60
weighted avg       1.00      1.00      1.00        60

□ Predicción para condiciones nuevas:
   Humedad: 80%
   Presión: 995 hPa
   ¿Lloverá?: Sí □
```



K VECINOS MÁS CERCANOS

22. K Vecinos Más Cercanos para Clasificación

En este ejercicio debes desarrollar una función que aplique el algoritmo de los k vecinos más cercanos (KNN) para un problema de clasificación.

Supongamos que tienes un conjunto de datos que contiene información sobre diferentes tipos de flores, y deseas predecir el tipo de flor en función de las características de pétalos y sépalos.

Utilizaremos el conjunto de datos Iris, que es un conjunto de datos de clasificación ampliamente utilizado en el aprendizaje automático.

```
def knn_clasificacion(datos, k=3):
```

```
# Ejemplo de uso con el conjunto de datos Iris
data = pd.read_csv('iris.csv')  # Reemplaza 'iris.csv' con tu archivo de datos
modelo_knn = knn_clasificacion(data, k=3)

# Estimaciones de clasificación para nuevas muestras
nuevas_muestras = pd.DataFrame({
    'LargoSepalo': [5.1, 6.0, 4.4],
    'AnchoSepalo': [3.5, 2.9, 3.2],
    'LargoPetalo': [1.4, 4.5, 1.3],
    'AnchoPetalo': [0.2, 1.5, 0.2]
})

estimaciones_clasificacion = modelo_knn.predict(nuevas_muestras)
print("Estimaciones de Clasificación:")
print(estimaciones_clasificacion)
```

Resultados:

```
Estimaciones de Clasificación:
['setosa' 'versicolor' 'setosa']
```

Objetivo de aprendizaje

23. Seguridad en Dispositivos IoT

Seguridad en Dispositivos IoT con k-NN

□ Contexto

Imagina que trabajas como **ingeniero de ciberseguridad** en una empresa que protege redes de dispositivos IoT. Tu equipo ha identificado que algunos dispositivos pueden representar un **riesgo** para la red debido a patrones de tráfico sospechosos.

Tu tarea es construir un **clasificador de seguridad** basado en **k-vecinos más cercanos (k-NN)** que permita identificar si un dispositivo IoT es **seguro** o **peligroso** basándose en su actividad de red.

□ Datos Disponibles

Para cada dispositivo conectado a la red, tenemos los siguientes datos:

- **paquetes_por_segundo** □ → Cantidad de paquetes enviados por segundo.
- **bytes_por_paquete** □ → Tamaño promedio de los paquetes en bytes.
- **protocolo** □ → Tipo de protocolo usado (1 = TCP, 2 = UDP, 3 = HTTP).
- **seguro** □□ → Clasificación (1 = seguro, 0 = peligroso).

Tu objetivo es usar estos datos para entrenar un modelo de **k-NN** que prediga si un nuevo

dispositivo IoT representa una amenaza.

□ Objetivos

1. **Generar datos sintéticos** □ con al menos 50 dispositivos IoT.
2. **Entrenar un modelo k-NN** □ usando `scikit-learn`.
3. **Evaluar el modelo** □ midiendo su precisión.
4. **Predecir la seguridad de un nuevo dispositivo IoT** □.

Recomendador de Personajes
□ "Recomendador de Personajes: ¿Qué tipo de personaje deberías elegir?"

□ Enunciado

En este ejercicio trabajarás como desarrollador de sistemas inteligentes para un nuevo videojuego tipo RPG online. El juego permite a los jugadores crear personajes y elegir entre distintos **roles o clases** (por ejemplo: guerrero, mago, arquero, curandero...).

Tu tarea es construir un **modelo de recomendación** que, dado un perfil de jugador (nivel, estilo de combate, número de partidas jugadas, etc.), recomiende **qué tipo de personaje** debería usar, basándose en datos históricos de otros jugadores similares.

□ Requerimientos

1. Crea una clase `Player` que represente a un jugador con los siguientes atributos:
    - `name`: nombre del jugador.
    - `level`: nivel del jugador (1 a 100).
    - `aggressiveness`: valor entre 0 y 1 que representa su estilo ofensivo.
    - `cooperation`: valor entre 0 y 1 que representa cuánto coopera con el equipo.

- `exploration`: valor entre 0 y 1 que representa cuánto le gusta explorar el mapa.
- `preferred_class`: clase de personaje que suele elegir (solo en los datos de entrenamiento).

2. Implementa un método `.to_features()` en la clase para convertir al jugador en una lista de características numéricas (sin la clase preferida).
3. Crea una clase `PlayerDataset` que contenga una lista de jugadores y proporcione:
   - `get_X()` → lista de listas de características.
   - `get_y()` → lista de clases preferidas.
4. Crea una clase `ClassRecommender` que use **KNN** para:
   - Entrenar el modelo a partir de un `PlayerDataset`.
   - Predecir la mejor clase para un nuevo jugador (`predict(player)`).
   - Obtener los `k` jugadores más parecidos (`get_nearest_neighbors(player)`).
5. (Opcional) Permite probar diferentes valores de `k` y evaluar la precisión del modelo con `cross_val_score`.

🧪 Ejemplo de uso

```
# Datos de entrenamiento
players = [
    Player("Alice", 20, 0.8, 0.2, 0.1, "Warrior"),
    Player("Bob", 45, 0.4, 0.8, 0.2, "Healer"),
    Player("Cleo", 33, 0.6, 0.4, 0.6, "Archer"),
    Player("Dan", 60, 0.3, 0.9, 0.3, "Healer"),
    Player("Eli", 50, 0.7, 0.2, 0.9, "Mage"),
    Player("Fay", 25, 0.9, 0.1, 0.2, "Warrior"),
]

# Nuevo jugador
new_player = Player("TestPlayer", 40, 0.6, 0.3, 0.8)

# Entrenamiento y predicción
dataset = PlayerDataset(players)
recommender = ClassRecommender(n_neighbors=3)
recommender.train(dataset)

# Resultado
recommended_class = recommender.predict(new_player)
neighbors_indices = recommender.get_nearest_neighbors(new_player)

print(f"Clase recomendada para {new_player.name}: {recommended_class}")
print("Jugadores similares:")
for i in neighbors_indices:
    print(f"- {players[i].name} ({players[i].preferred_class})")
```

🧾 Salida esperada

```
Clase recomendada para TestPlayer: Archer
Jugadores similares:
- Bob (Healer)
```

```
4. - Cleo (Archer)
5. - Eli (Mage)
```

24. Recomendador de Personajes
□ "Recomendador de Personajes: ¿Qué tipo de personaje deberías elegir?"

□ Enunciado

En este ejercicio trabajarás como desarrollador de sistemas inteligentes para un nuevo videojuego tipo RPG online. El juego permite a los jugadores crear personajes y elegir entre distintos **roles o clases** (por ejemplo: guerrero, mago, arquero, curandero...).

Tu tarea es construir un **modelo de recomendación** que, dado un perfil de jugador (nivel, estilo de combate, número de partidas jugadas, etc.), recomiende **qué tipo de personaje** debería usar, basándose en datos históricos de otros jugadores similares.

 Requerimientos

1. Crea una clase `Player` que represente a un jugador con los siguientes atributos:
   - `name`: nombre del jugador.
   - `level`: nivel del jugador (1 a 100).
   - `aggressiveness`: valor entre 0 y 1 que representa su estilo ofensivo.
   - `cooperation`: valor entre 0 y 1 que representa cuánto coopera con el equipo.
   - `exploration`: valor entre 0 y 1 que representa cuánto le gusta explorar el mapa.
   - `preferred_class`: clase de personaje que suele elegir (solo en los datos de entrenamiento).
2. Implementa un método `.to_features()` en la clase para convertir al jugador en una lista de características numéricas (sin la clase preferida).
3. Crea una clase `PlayerDataset` que contenga una lista de jugadores y proporcione:
   - `get_X()` → lista de listas de características.
   - `get_y()` → lista de clases preferidas.
4. Crea una clase `ClassRecommender` que use **KNN** para:
   - Entrenar el modelo a partir de un `PlayerDataset`.
   - Predecir la mejor clase para un nuevo jugador (`predict(player)`).
   - Obtener los `k` jugadores más parecidos (`get_nearest_neighbors(player)`).
5. (Opcional) Permite probar diferentes valores de `k` y evaluar la precisión del modelo con `cross_val_score`.

 Ejemplo de uso

```
# Datos de entrenamiento
players = [
    Player("Alice", 20, 0.8, 0.2, 0.1, "Warrior"),
    Player("Bob", 45, 0.4, 0.8, 0.2, "Healer"),
    Player("Cleo", 33, 0.6, 0.4, 0.6, "Archer"),
    Player("Dan", 60, 0.3, 0.9, 0.3, "Healer"),
    Player("Eli", 50, 0.7, 0.2, 0.9, "Mage"),
    Player("Fay", 25, 0.9, 0.1, 0.2, "Warrior"),
]

# Nuevo jugador
new_player = Player("TestPlayer", 40, 0.6, 0.3, 0.8)

# Entrenamiento y predicción
dataset = PlayerDataset(players)
recommender = ClassRecommender(n_neighbors=3)
recommender.train(dataset)

```

```
# Resultado
recommended_class = recommender.predict(new_player)
neighbors_indices = recommender.get_nearest_neighbors(new_player)

print(f"Clase recomendada para {new_player.name}: {recommended_class}")
print("Jugadores similares:")
for i in neighbors_indices:
    print(f"- {players[i].name} ({players[i].preferred_class})")
```

Salida esperada

```
Clase recomendada para TestPlayer: Archer
Jugadores similares:
- Bob (Healer)
- Cleo (Archer)
- Eli (Mage)
```

25. Recomendador de canciones inteligente

Contexto

Estás desarrollando un sistema para una plataforma musical que quiere ofrecer **recomendaciones automáticas** basadas en características cuantitativas de las canciones, como su energía o duración.

Utilizarás el algoritmo **K-Nearest Neighbors (KNN)** de la biblioteca `scikit-learn` para encontrar las canciones más similares a una canción objetivo.

□ Objetivo del ejercicio

Implementar un sistema de recomendación de canciones en Python, usando el modelo de K Vecinos Más Cercanos de `scikit-learn`.

El sistema debe permitir recomendar canciones similares a partir de características musicales numéricas.

□ Requisitos

□ 1. Clase `Song`

Crea una clase `Song` que represente una canción, con los siguientes atributos:

- `title` (str): título de la canción.
- `artist` (str): artista o grupo musical.
- `energy` (float): energía de la canción (0.4 a 1.0).
- `danceability` (float): cuán bailable es la canción (0.4 a 1.0).
- `duration` (int): duración en segundos (180 a 300).
- `popularity` (int): nivel de popularidad (50 a 100).

La clase debe incluir:

- Un método `to_vector()` que devuelva una lista con los valores `[energy, danceability, duration, popularity]`.
- Un método `__str__()` que permita imprimir la canción en formato `"Song Title by Artist"`.

□ 2. Clase `SongRecommender`

Crea una clase `SongRecommender` que use el algoritmo de KNN de `scikit-learn`:

- El constructor debe aceptar un parámetro `k` (número de vecinos a considerar).
- El método `fit(song_list)` debe:
  - Convertir la lista de canciones en una matriz de características numéricas.
  - Ajustar el modelo `NearestNeighbors` con estas características.
- El método `recommend(target_song)` debe:

- Obtener los `k` vecinos más cercanos a la canción objetivo.
- Devolver la lista de canciones recomendadas (sin incluir la propia canción objetivo si aparece).

🧩 3. Clase `SongGenerator`

Crea una clase `SongGenerator` con:

- Un parámetro `num_songs` (por defecto 30).
- Un método `generate()` que genere canciones aleatorias con `numpy`, usando nombres como `"Song1"`, `"Song2"`, etc., y artistas `"Artist1"`, `"Artist2"`, etc.

🧩 4. Clase `SongRecommendationExample`

Crea una clase de ejemplo que:

- Genere una lista de canciones con `SongGenerator`.
- Defina una canción personalizada como objetivo (`target_song`).
- Cree una instancia de `SongRecommender`, la entrene con las canciones y obtenga recomendaciones.
- Imprima por pantalla las canciones recomendadas.

Ejemplo de salida:

```
example = SongRecommendationExample()
example.run()
```

Salida esperada

```
🎵 Recomendaciones para 'Mi Canción':
 - Song29 by Artist4
 - Song11 by Artist1
 - Song25 by Artist5
```

🧠 Recomendaciones para completar el ejercicio

- Usa `numpy` para generar valores aleatorios.
- Recuerda importar `NearestNeighbors` desde `sklearn.neighbors`.
- Asegúrate de convertir los objetos `Song` a vectores antes de ajustar o predecir con el modelo.
- No incluyas la canción objetivo entre las recomendaciones (verifica si es necesario).

26. Clasificador inteligente de materiales reciclables
Contexto:
Imagina que trabajas para una empresa de reciclaje inteligente.

Tu tarea consiste en diseñar un sistema que pueda **predecir automáticamente** si un objeto es **plástico, metal o papel**, a partir de sus propiedades físicas, usando el algoritmo de **k vecinos más cercanos (KNN)**.

Vas a utilizar Python con las librerías `numpy`, `pandas`, `matplotlib` y `sklearn` para entrenar y visualizar el modelo.

□ Objetivo

Implementa las siguientes clases:

1. `RecyclableItem`

Representa un objeto reciclable con tres atributos:

- `weight`: peso del objeto en gramos.
- `volume`: volumen en cm^3.
- `material_type`: tipo de material codificado como:
  - `0` para **plástico**
  - `1` para **metal**
  - `2` para **papel**

Método necesario:

- `to_vector(self)`: devuelve `[weight, volume]`, útil para alimentar el modelo.

2. `RecyclableDataGenerator`

Genera objetos sintéticos para entrenar el modelo.

Constructor:

- `num_samples`: número total de objetos a generar (repartidos entre los tres tipos de material).

Método:

- `generate(self)`: genera y devuelve una lista de objetos `RecyclableItem` con las siguientes características:
  - **Plástico (0):**
    - Peso: entre 80 y 120 g
    - Volumen: entre 90 y 130 cm^3
  - **Metal (1):**
    - Peso: entre 180 y 220 g
    - Volumen: entre 70 y 110 cm^3
  - **Papel (2):**

- Peso: entre 40 y 70 g
- Volumen: entre 120 y 160 cm^3

3. `RecyclableMaterialClassifier`

Clasificador que entrena un modelo de KNN.

Métodos:

- `fit(records)`: entrena el modelo con una lista de objetos `RecyclableItem`.
- `predict(weight, volume)`: devuelve el tipo de material predicho (`0`, `1` o `2`) para un nuevo objeto.
- `evaluate(records)`: imprime métricas de clasificación (`classification_report`, `confusion_matrix`) con un conjunto de prueba.

4. `RecyclablePredictionExample`

Clase que coordina todo el flujo:

- Genera los datos.
- Separa en entrenamiento y prueba.
- Entrena el clasificador.
- Evalúa el rendimiento.
- Hace una **predicción para un nuevo objeto** (por ejemplo, peso = 110, volumen = 105).
- Visualiza los datos y las predicciones en un gráfico 2D con colores distintos para cada tipo de material.

□ Ejemplo de uso

```
example = RecyclablePredictionExample()
example.run()
```

Salida esperada

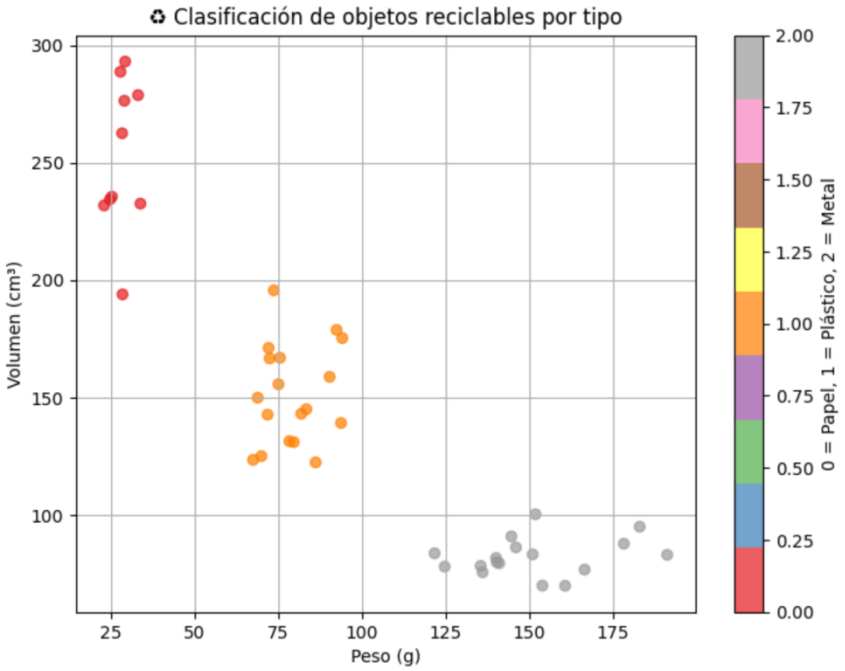
```
[[10  0  0]
```

```
[ 0 18  0]
[ 0  0 17]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00        18
           2       1.00      1.00      1.00        17

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45


 Predicción para un nuevo objeto:
   Peso: 60g, Volumen: 180cm³
   Tipo estimado: Plástico
```



ÁRBOLES DE DECISIÓN

27. Arboles de decisión
Implementar una función en Python que entrene un **árbol de decisión** y lo use para hacer predicciones en un conjunto de datos.

Instrucciones

1. Implementa una función llamada `entrenar_arbol_decision(X_train, y_train, X_test)` que:
   - Entrene un árbol de decisión con los datos de entrenamiento `X_train` y `y_train`.
   - Prediga los valores de `X_test`.
   - Devuelva un array con las predicciones.
2. Usa `DecisionTreeClassifier` de `sklearn.tree` con `random_state=42` para asegurar reproducibilidad.
3. **No modifiques los parámetros de entrenamiento** (como la profundidad del árbol o la función de división).
4. **Prueba la función con un conjunto de datos real** como el conjunto de datos de flores **Iris**.

Ejemplo de Uso

Tu función debe funcionar correctamente con este código de prueba:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np

# Cargar el dataset de flores Iris
iris = load_iris()
X = iris.data  # Características
y = iris.target  # Clases de las flores (Setosa, Versicolor, Virginica)

# Dividir en conjunto de entrenamiento y prueba (80%-20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Llamar a la función que debes implementar
predicciones = entrenar_arbol_decision(X_train, y_train, X_test)

# Mostrar algunas predicciones
print("Predicciones del Árbol de Decisión:", predicciones[:10])
print("Valores reales:                    ", y_test[:10])
```

28. Métricas en árboles de decisión
El objetivo de este ejercicio es que los estudiantes implementen una función que:

1. **Entrene un árbol de decisión** usando `DecisionTreeClassifier` de `sklearn`.
2. **Haga predicciones** en un conjunto de prueba.
3. **Evalúe el modelo** utilizando métricas como **precisión (accuracy)**, **matriz de confusión** y **reporte de clasificación**.
4. **Pase pruebas unitarias (`unittest`)** que validen el funcionamiento correcto del código.

Instrucciones

1. Implementa una función llamada `entrenar_y_evaluar_arbol(X_train, y_train, X_test, y_test)` que:
    - **Entrene** un modelo `DecisionTreeClassifier` con los datos de entrenamiento (`X_train`, `y_train`).
    - **Prediga** los valores de `X_test`.
    - **Evalúe el modelo** usando:
        - Precisión (`accuracy_score`)
        - Matriz de confusión (`confusion_matrix`)
        - Reporte de clasificación (`classification_report`)
    - Devuelva un diccionario con:
        - `predicciones`: Un array con las predicciones del modelo.
        - `accuracy`: Un número flotante con la precisión.
        - `matriz_confusion`: Una matriz de confusión.
        - `reporte`: Un string con el reporte de clasificación.
2. **Usa `random_state=42` en `DecisionTreeClassifier`** para reproducibilidad.
3. **Prueba la función con el dataset Iris**, asegurando que el modelo tenga al menos **85% de precisión** en los datos de prueba.

Ejemplo de Uso

Una vez implementada la función, debe ejecutarse correctamente con este código de prueba:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np

# Cargar el dataset de flores Iris
iris = load_iris()
X = iris.data  # Características
y = iris.target  # Clases de las flores (Setosa, Versicolor, Virginica)

# Dividir en conjunto de entrenamiento (80%) y prueba (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

```

```python
# Importar la función implementada
from solution import entrenar_y_evaluar_arbol

# Llamar a la función y obtener las métricas
resultados = entrenar_y_evaluar_arbol(X_train, y_train, X_test, y_test)

# Mostrar los resultados
print("Precisión del modelo:", resultados["accuracy"])
print("Matriz de Confusión:\n", resultados["matriz_confusion"])
print("Reporte de Clasificación:\n", resultados["reporte"])
```

Salida esperada (aproximada):

```
Precisión del modelo: 1.0
Matriz de Confusión:
 [[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
Reporte de Clasificación:
              precision    recall  f1-score   support

      Setosa       1.00      1.00      1.00        10
  Versicolor       1.00      1.00      1.00         9
   Virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

29. Recomendador de plataforma
 Título:

"¿Web, móvil o escritorio? – Recomendador de plataforma"

 Enunciado:

Imagina que trabajas en una **consultora tecnológica** que ayuda a startups y pequeñas empresas a decidir **qué tipo de plataforma** es la más adecuada para sus aplicaciones: **web, móvil o escritorio**.

Tu tarea será entrenar un modelo de aprendizaje automático que, **dado un conjunto de características de un proyecto**, sea capaz de predecir automáticamente **la plataforma recomendada**.

 Datos de entrada (proyectos)

Cada proyecto tiene las siguientes características:

Atributo	Tipo	Descripción
`project_name`	str	Nombre del proyecto
`team_size`	int	Número de personas en el equipo
`budget`	float	Presupuesto disponible en miles de euros
`duration_months`	int	Duración estimada del proyecto
`realtime_required`	bool	¿Se necesita comunicación en tiempo real?
`needs_offline`	bool	¿Debe funcionar sin conexión a internet?
`target_users`	str	Público objetivo: `"global"`, `"local"`, `"empresa"`
`recommended_platform`	str	`"web"`, `"mobile"`, `"desktop"` *(solo en los proyectos de entrenamiento)*

 Objetivo

Tu modelo deberá predecir el valor de `recommended_platform` para un nuevo proyecto a partir de sus características.

 Requisitos del ejercicio

1. **Crear una clase** `Project` para representar cada proyecto.

2. **Crear una clase** `ProjectDataset` que contenga una lista de proyectos y permita extraer los datos necesarios para el modelo.
3. **Crear una clase** `PlatformRecommender` que entrene un modelo basado en **árboles de decisión** (`DecisionTreeClassifier`) y permita hacer predicciones.

🔹 Ejemplo de uso

```
projects = [
    Project("AppGlobal", 5, 25.0, 6, True, False, "global", "web"),
    Project("IntranetCorp", 10, 40.0, 12, False, True, "empresa", "desktop"),
    Project("LocalDelivery", 3, 20.0, 4, True, True, "local", "mobile"),
    Project("CloudDashboard", 6, 50.0, 8, True, False, "empresa", "web"),
    Project("OfflineTool", 4, 15.0, 6, False, True, "local", "desktop"),
    Project("SocialBuzz", 2, 10.0, 3, True, False, "global", "mobile"),
]

new_project = Project("AIChatApp", 4, 30.0, 5, True, False, "global")

dataset = ProjectDataset(projects)
recommender = PlatformRecommender()
recommender.train(dataset)

prediction = recommender.predict(new_project)
print(f"Plataforma recomendada: {prediction}")
```

🔹 Salida esperada

```
Plataforma recomendada: mobile
```

🔹 Requisitos adicionales

- Puedes usar `LabelEncoder` para transformar variables categóricas (`target_users`).
- Asegúrate de convertir los booleanos (`realtime_required`, `needs_offline`) en enteros (0 o 1) antes de entrenar el modelo.
- Evalúa tu modelo con diferentes ejemplos para ver cómo se comporta.

30. Clasificador de snacks saludables
Objetivo:

En este ejercicio, aprenderás a crear un clasificador para predecir si un snack es saludable o no, basándote en características nutricionales como las calorías, azúcar, proteínas, grasas y fibra.

Usaremos un árbol de decisión para crear un modelo que prediga si un snack es saludable en función de estos atributos.

Descripción:

Imagina que trabajas en una aplicación de salud que recomienda snacks a los usuarios. Tienes acceso a un conjunto de datos que contiene información sobre varios snacks y su contenido nutricional.

Usaremos estos datos para entrenar un modelo que pueda predecir si un snack es saludable basándose en sus atributos.

Pasos a seguir:

1. **Creación de la clase `Snack`:**
   - Define una clase `Snack` que tenga los siguientes atributos: `calories`, `sugar`, `protein`, `fat`, `fiber`, y un atributo opcional `is_healthy`, que será el resultado que queremos predecir (1 si el snack es saludable, 0 si no lo es).
   - Crea un método `to_vector()` que convierta un snack en un vector de características (calorías, azúcar, proteínas, grasas, fibra).
2. **Generación de Datos Sintéticos con la clase `SnackGenerator`:**
   - Crea una clase `SnackGenerator` que sea capaz de generar un conjunto de datos sintéticos con snacks. Esta clase debe crear entre 50 y 200 snacks con valores aleatorios para las características mencionadas.
   - La variable `is_healthy` debe seguir una regla aproximada: un snack es saludable si tiene menos de 200 calorías, menos de 15 gramos de azúcar, menos de 10 gramos de grasa, y al menos 5 gramos de proteína o fibra.
3. **Clasificador de Snacks con Árbol de Decisión:**
   - Crea una clase `SnackClassifier` que use un árbol de decisión para clasificar los snacks.
   - Esta clase debe tener dos métodos:
     - `fit()`: entrenar el modelo usando un conjunto de snacks y sus etiquetas (`is_healthy`).
     - `predict()`: predecir si un snack específico es saludable o no.
4. **Crear un Ejemplo de Uso:**
   - Crea un objeto de la clase `SnackRecommendationExample` que entrene el clasificador utilizando el generador de snacks.

- Luego, crea un snack de prueba con valores nutricionales conocidos, como 150 calorías, 10 gramos de azúcar, 6 gramos de proteína, 5 gramos de grasa y 3 gramos de fibra.
- Usa el clasificador para predecir si este snack es saludable y muestra la predicción.

Requisitos:

- **Uso de Árbol de Decisión:** Para realizar la clasificación, usa la librería `sklearn` y su `DecisionTreeClassifier`.
- **Generación de datos:** Usa `numpy` para generar valores aleatorios.
- **Impresión de resultados:** Imprime la información nutricional del snack de prueba junto con la predicción de si es saludable o no.

Resultado esperado:

Al ejecutar el código, el sistema debe mostrar la información nutricional del snack de prueba y una predicción indicando si es saludable o no.

Ejemplo de uso

```
# Ejecutar ejemplo
example = SnackRecommendationExample()
example.run()
```

Salida esperada

```
 Snack Info:
Calories: 150, Sugar: 10g, Protein: 6g, Fat: 5g, Fiber: 3g
 Predicción: Este snack no es saludable.
```

31. Clasificador de jugadores de baloncesto
Objetivo del ejercicio:

Tu misión es construir un modelo inteligente que clasifique a jugadores de baloncesto según su rendimiento en tres categorías: **"Bajo"**, **"Medio"** y **"Alto"**, utilizando para ello sus características físicas y estadísticas de juego.

Usarás el algoritmo de **árboles de decisión** junto con `NumPy`, `pandas`, `matplotlib` y `scikit-learn`.

Contexto del problema

Un equipo de baloncesto ficticio está evaluando a nuevos jugadores y necesita una herramienta que, a partir de la **altura**, el **peso** y el **promedio de puntos por partido**, determine automáticamente el nivel de rendimiento del jugador.

Esta herramienta será clave para seleccionar a los mejores candidatos.

Estructura sugerida de la solución

1. `BasketballPlayer`

Una clase que representa a cada jugador. Sus atributos son:

- `height` (int): altura en centímetros.
- `weight` (int): peso en kilogramos.
- `avg_points` (float): promedio de puntos por partido.
- `performance` (str): nivel de rendimiento, con valores `"Bajo"`, `"Medio"` o `"Alto"`.

Método útil:

- `to_vector()`: devuelve `[height, weight, avg_points]` para ser usado por el modelo.

2. `BasketballDataGenerator`

Una clase que genera datos sintéticos simulando jugadores reales.

Método clave:

- `generate()`: devuelve una lista de objetos `BasketballPlayer`. La clasificación se basa en el promedio de puntos:
  - Menos de 8 puntos → **"Bajo"**
  - Entre 8 y 15 puntos → **"Medio"**

- Más de 15 puntos → **"Alto"**

3. `BasketballPerformanceClassifier`

Encapsula el modelo de árbol de decisión. Métodos clave:

- `fit(players)`: entrena el modelo con una lista de jugadores.
- `predict(height, weight, avg_points)`: predice el rendimiento de un nuevo jugador.
- `evaluate(players)`: imprime la matriz de confusión y el informe de clasificación sobre un conjunto de prueba.

4. `BasketballPredictionExample`

Contiene el flujo principal:

- Generar datos.
- Dividirlos en entrenamiento y prueba.
- Entrenar y evaluar el clasificador.
- Hacer una predicción para un nuevo jugador (por ejemplo: altura = 198 cm, peso = 92 kg, puntos = 17).
- Visualizar los jugadores usando `matplotlib`, diferenciando el rendimiento por colores.

□ Visualización esperada

Un gráfico de dispersión donde cada punto representa un jugador.

El eje X muestra la **altura** y el eje Y el **promedio de puntos**.

El color indica el rendimiento:

- □ Bajo
- □ Medio
- □ Alto

□ Requisitos técnicos

- Usa `NumPy` para generar datos aleatorios.
- Usa `pandas` para crear el DataFrame de visualización.
- Usa `DecisionTreeClassifier` de `sklearn.tree`.
- Representa visualmente los datos con `matplotlib`.

□ Ejemplo de uso

```
example = BasketballPredictionExample()
example.run()
```

Salida esperada

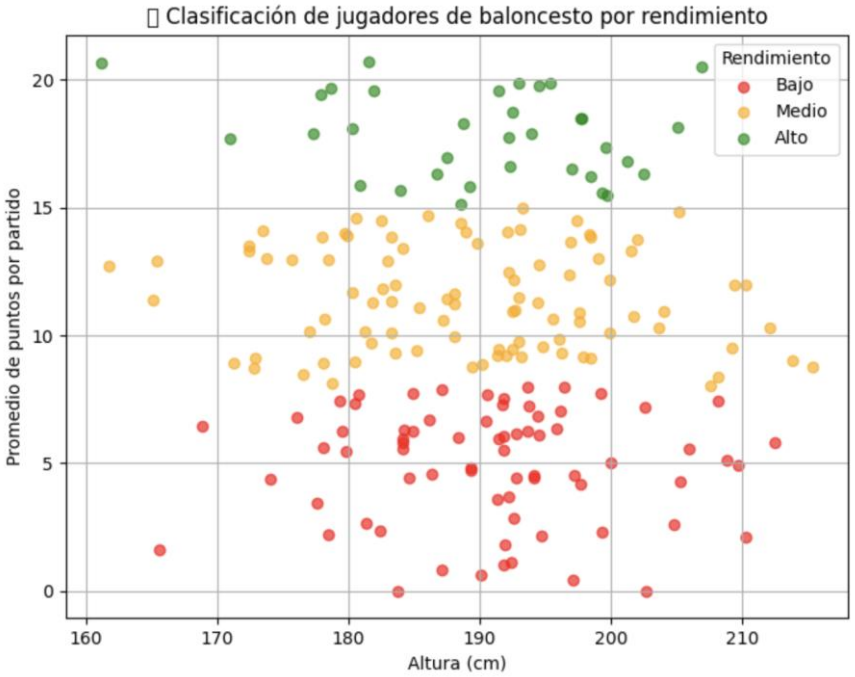
```
Confusion Matrix:
[[10  0  0]
 [ 0 23  0]
 [ 0  0 27]]

Classification Report:
              precision    recall  f1-score   support

        Alto       1.00      1.00      1.00        10
       Bajo       1.00      1.00      1.00        23
      Medio       1.00      1.00      1.00        27

    accuracy                           1.00        60
   macro avg       1.00      1.00      1.00        60
weighted avg       1.00      1.00      1.00        60


□ Predicción personalizada → Altura: 198 cm, Peso: 92 kg, Prom. puntos: 17
   → Categoría predicha: Alto
```



RANDOM FOREST

32. Random Forest
El objetivo es implementar una función que:

1. **Entrene un modelo Random Forest** (`RandomForestClassifier`).
2. **Haga predicciones** en datos de prueba.
3. **Evalúe el rendimiento del modelo** con:
    - **Precisión (`accuracy_score`)**
    - **Matriz de confusión (`confusion_matrix`)**
    - **Reporte de clasificación (`classification_report`)**
4. **Devuelva los resultados en un diccionario**.
5. **Supervise la implementación con pruebas unitarias (`unittest`)**.

Instrucciones

1. Implementa una función llamada `entrenar_y_evaluar_random_forest(X_train, y_train, X_test, y_test)` que:
    - Entrene un `RandomForestClassifier(n_estimators=100, random_state=42)`.
    - Prediga los valores de `X_test`.
    - Calcule las métricas de evaluación mencionadas.
    - Devuelva un diccionario con:
        - `"predicciones"`: Array de predicciones del modelo.
        - `"accuracy"`: Precisión del modelo en los datos de prueba.
        - `"matriz_confusion"`: Matriz de confusión.
        - `"reporte"`: Reporte de clasificación.
2. **Usa el dataset de vinos (`wine dataset`)** de `sklearn.datasets`.
3. **Asegúrate de que el modelo tenga al menos 90% de precisión** en los datos de prueba.

Ejemplo de Uso

El siguiente código debería ejecutarse correctamente una vez que implementes la función:

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
import numpy as np

# Cargar el dataset de vinos
wine = load_wine()
X = wine.data  # Características
y = wine.target  # Clases de vinos
```

```
9.
10. # Dividir en conjunto de entrenamiento (80%) y prueba (20%)
11. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
12.
13. # Importar la función implementada
14. from solution import entrenar_y_evaluar_random_forest
15.
16. # Llamar a la función y obtener las métricas
17. resultados = entrenar_y_evaluar_random_forest(X_train, y_train, X_test,
    y_test)
18.
19. # Mostrar los resultados
20. print("Precisión del modelo:", resultados["accuracy"])
21. print("Matriz de Confusión:\n", resultados["matriz_confusion"])
22. print("Reporte de Clasificación:\n", resultados["reporte"])
```

Salida esperada (aproximada)

```
1. Precisión del modelo: 1.0
2. Matriz de Confusión:
3.  [[14  0  0]
4.  [ 0 14  0]
5.  [ 0  0  8]]
6. Reporte de Clasificación:
7.                precision    recall  f1-score   support
8.
9.      Clase 0       1.00      1.00      1.00        14
10.     Clase 1       1.00      1.00      1.00        14
11.     Clase 2       1.00      1.00      1.00         8
12.
13.    accuracy                           1.00        36
14.   macro avg       1.00      1.00      1.00        36
15. weighted avg      1.00      1.00      1.00        36
```

33. Recomendador de lenguajes de programación
 Ejercicio:

¿Qué lenguaje de programación es ideal para este proyecto?

 Contexto:

Imagina que una empresa tecnológica está analizando diversos proyectos para decidir con qué lenguaje de programación desarrollarlos. Según ciertas características del proyecto (velocidad requerida, facilidad de mantenimiento, disponibilidad de bibliotecas, tipo de aplicación, y rendimiento deseado), deben predecir si el lenguaje ideal es **Python**, **JavaScript**, **Java** o **C++**.

 Enunciado del ejercicio:

Tu tarea es construir un modelo de clasificación usando el algoritmo **Random Forest** que, dado un conjunto de características de un proyecto tecnológico, prediga cuál es el **lenguaje de programación más adecuado**.

 Parte 1: Preparar los datos

Los datos se representan como arrays de NumPy. Cada proyecto tiene estas características:

Atributo	Tipo	Descripción
velocidad_requerida	float (0-1)	¿Qué tan importante es la velocidad de ejecución?
facilidad_mantenimiento	float (0-1)	¿Qué tan fácil debe ser mantener el código?
disponibilidad_libs	float (0-1)	¿Qué tan importante es que haya muchas librerías?
tipo_app	int (0-2)	0: web, 1: escritorio, 2: sistema embebido
rendimiento_deseado	float (0-1)	¿Qué tan importante es el rendimiento general?

La **etiqueta** será un número entero que representa el lenguaje:

- 0 → Python
- 1 → JavaScript
- 2 → Java
- 3 → C++

□ Parte 2: Crear el modelo

1. Genera un conjunto de datos de ejemplo (puedes crear 100 proyectos artificiales con NumPy).
2. Entrena un modelo de **Random Forest** (`sklearn.ensemble.RandomForestClassifier`).
3. Implementa una función que reciba un array con las 5 características de un proyecto nuevo y devuelva el lenguaje más adecuado.

□ Objetivo final:

Implementa una clase llamada `LanguagePredictor` que tenga:

```
class LanguagePredictor:
    def __init__(self):
        ...

    def train(self, X, y):
        ...

    def predict(self, features: np.ndarray) -> str:
        ...
```

□ Sugerencia para el alumno:

- Usa `numpy.random.rand` para generar las características (entre 0 y 1).
- Usa `numpy.random.randint` para crear el tipo de app (0, 1 o 2).
- Genera etiquetas de forma algo correlacionada: por ejemplo, si `rendimiento_deseado` > 0.8 y tipo_app == 2, tal vez el lenguaje sea C++.
- Usa un diccionario para traducir el resultado del modelo (número) al lenguaje de programación.

□ 3. Ejemplo de uso

```
# Generar datos y entrenar
X, y = generate_dataset()
predictor = LanguagePredictor()
predictor.train(X, y)

# Crear un proyecto nuevo
new_project = np.array([0.7, 0.9, 0.5, 1, 0.6])  # Características del proyecto

# Predecir lenguaje ideal
pred = predictor.predict(new_project)
```

```
print(f"Lenguaje recomendado para el nuevo proyecto: {pred}")
```

 Salida esperada:

```
Lenguaje recomendado para el nuevo proyecto: JavaScript
```

34. Recomendador de videojuegos
 Descripción:

Vas a construir un **sistema de recomendación de videojuegos** que pueda predecir si a un jugador le gustará o no un videojuego basándose en características como la acción, la estrategia, los gráficos o la dificultad.

Para ello, utilizarás:

- Datos sintéticos generados con `numpy`
- Un modelo de clasificación usando **Random Forest** de `sklearn`

 Objetivo:

1. Crear una clase `VideoGame` que represente un videojuego con características numéricas.
2. Generar una lista de videojuegos con etiquetas (le gusta/no le gusta) usando reglas sencillas.
3. Entrenar un modelo con **RandomForestClassifier**.
4. Usar el modelo para predecir si un nuevo videojuego será del gusto de un jugador.

 Especificaciones del ejercicio:

1. Crea una clase `VideoGame` con los siguientes atributos:
   - `action` (nivel de acción, de 0 a 1)
   - `strategy` (nivel de estrategia, de 0 a 1)
   - `graphics` (calidad gráfica, de 0 a 1)
   - `difficulty` (nivel de dificultad, de 0 a 1)
   - `liked` (opcional: 1 si le gusta al jugador, 0 si no)
2. Crea una clase `VideoGameGenerator` que genere datos sintéticos de videojuegos, incluyendo si fueron o no del gusto de los jugadores (campo `liked`).

   Tip: Una regla simple para considerar que un juego gustó puede ser:

   a. `liked = int((action > 0.7 or graphics > 0.7) and difficulty < 0.7)`

3. Crea la clase `VideoGameClassifier` que:
   - Entrene un **Random Forest** con los videojuegos generados.
   - Pueda predecir si le gustará un nuevo videojuego al jugador.

4. Crea una clase de ejemplo `VideoGameRecommendationExample` donde:
    - Generas 100 videojuegos aleatorios para entrenar.
    - Predices si al jugador le gustará un nuevo juego con:
        - `new_game = VideoGame(action=0.9, strategy=0.4, graphics=0.8, difficulty=0.3)`

Ejemplo de uso

```
example = VideoGameRecommendationExample()
example.run()
```

Salida esperada

```
 Nuevo juego:
Action: 0.9, Strategy: 0.4, Graphics: 0.8, Difficulty: 0.3
 Le gustará al jugador el juego? Si!
```

35. Predicción del nivel de estrés
□ Contexto

Los niveles de estrés afectan directamente a la salud física y mental.

En este proyecto, trabajarás como si fueras parte del equipo de desarrollo de un sistema de monitoreo de estrés para deportistas de alto rendimiento o trabajadores en ambientes exigentes.

Se te ha encomendado diseñar un clasificador que, a partir de tres medidas fisiológicas, pueda **predecir el nivel de estrés de una persona**.

Para ello, deberás simular datos realistas, entrenar un modelo de aprendizaje automático y visualizar los resultados.

□ Objetivos

1. **Simular datos fisiológicos** (ritmo cardíaco, nivel de cortisol y conductancia de la piel).
2. **Clasificar el nivel de estrés** de las personas como:
   □ Bajo, □ Moderado o □ Alto.
3. **Entrenar un clasificador Random Forest**.
4. **Evaluar el rendimiento** del modelo.
5. **Realizar predicciones personalizadas**.
6. **Visualizar los datos y resultados** con gráficos interpretables.

□ Requisitos Técnicos

Debes usar:

- `NumPy` para generar datos.
- `Pandas` para manipular estructuras.
- `matplotlib.pyplot` para visualizar.
- `sklearn` para entrenamiento del modelo y métricas.
- Programación orientada a objetos (clases bien definidas).

□□ Parte 1: Clase para representar individuos

Crea una clase llamada `Individual` con los siguientes atributos:

- Ritmo cardíaco (`heart_rate`) en pulsaciones por minuto.
- Nivel de cortisol (`cortisol_level`) en µg/dL.
- Conductancia de la piel (`skin_conductance`) en µS.
- Nivel de estrés (`stress_level`): cadena de texto (`'Bajo'`, `'Moderado'` o `'Alto'`).

Incluye un método `to_vector()` que devuelva solo las tres primeras variables como lista.

□ Parte 2: Simulador de datos

Crea una clase `StressDataGenerator` que genere una lista de objetos `Individual` con valores aleatorios realistas:

- Ritmo cardíaco: media 75, desviación estándar 15.
- Cortisol: media 12, desviación estándar 4.
- Conductancia: media 5, desviación estándar 1.5.

Clasifica los individuos según estas reglas:

- □ Alto: si **cualquiera** de las tres medidas supera estos umbrales:
  - Ritmo cardíaco > 90
  - Cortisol > 18
  - Conductancia > 6.5
- □ Moderado: si **alguna** supera:
  - Ritmo cardíaco > 70
  - Cortisol > 10
  - Conductancia > 4.5

  pero no cumple los criterios de "Alto".
- □ Bajo: si ninguna medida supera esos valores.

□ Parte 3: Clasificador con Random Forest

Crea una clase `StressClassifier` con los métodos:

- `fit(individuals)` → entrena el modelo con datos.
- `predict(heart_rate, cortisol, conductance)` → devuelve el nivel de estrés estimado.
- `evaluate(test_data)` → imprime matriz de confusión e informe de clasificación.

□ Parte 4: Prueba y predicción personalizada

1. Genera 300 datos simulados.
2. Divide el conjunto en entrenamiento y prueba (70%-30%).
3. Evalúa el modelo con métricas de `sklearn`.
4. Predice el estrés de un caso con valores como:

- Ritmo cardíaco: 95
- Cortisol: 20
- Conductancia: 7

□ Parte 5: Visualización de los datos

1. Usa `pandas` para convertir los datos simulados a un `DataFrame`.
2. Genera un gráfico de dispersión con `matplotlib`:
   - Eje X: nivel de cortisol
   - Eje Y: ritmo cardíaco
   - Color de los puntos según el nivel de estrés

Usa colores representativos:

- Verde → Bajo
- Naranja → Moderado
- Rojo → Alto

Agrega título, leyenda y cuadrícula.

□ Ejemplo de uso

```
example = StressAnalysisExample()
example.run()
```

Salida esperada

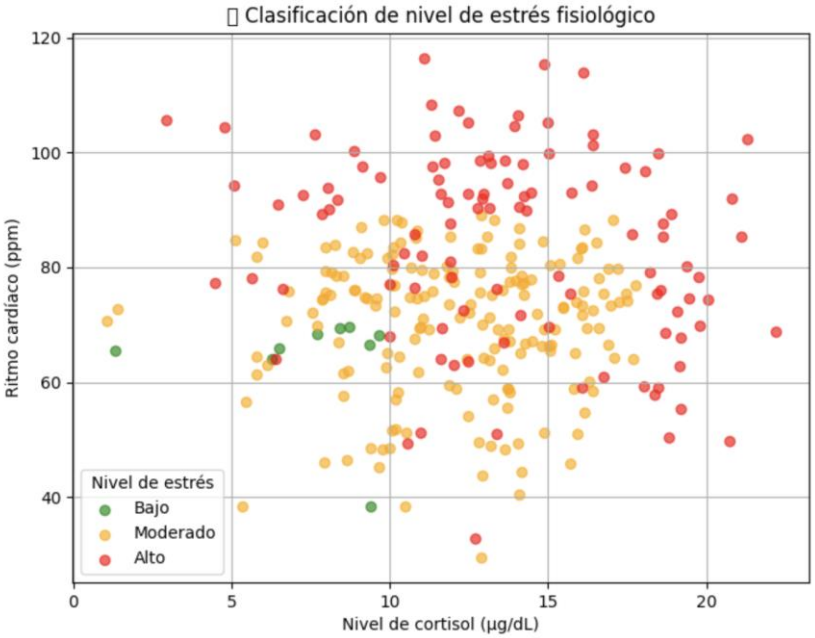
```
□ Matriz de confusión:
[[33  0  0]
 [ 0  2  1]
 [ 0  0 54]]

□ Informe de clasificación:
              precision    recall  f1-score   support

        Alto       1.00      1.00      1.00        33
        Bajo       1.00      0.67      0.80         3
    Moderado       0.98      1.00      0.99        54

    accuracy                           0.99        90
   macro avg       0.99      0.89      0.93        90
weighted avg       0.99      0.99      0.99        90


□ Predicción para individuo personalizado:
  Ritmo cardíaco: 95, Cortisol: 20, Conductancia: 7
  → Nivel estimado de estrés: Alto
```


Clasificación de nivel de estrés fisiológico
Ritmo cardíaco (ppm)
Nivel de cortisol (µg/dL)
Nivel de estrés
Bajo
Moderado
Alto
120
100
80
60
40
0
5
10
15
20

MÁQUINAS DE VECTORES DE SOPORTE

36. Recomendador de lenguajes de programación
□ Ejercicio:

¿Qué lenguaje de programación es ideal para este proyecto?

□ Contexto:

Imagina que una empresa tecnológica está analizando diversos proyectos para decidir con qué lenguaje de programación desarrollarlos. Según ciertas características del proyecto (velocidad requerida, facilidad de mantenimiento, disponibilidad de bibliotecas, tipo de aplicación, y rendimiento deseado), deben predecir si el lenguaje ideal es **Python**, **JavaScript**, **Java** o **C++**.

□ Enunciado del ejercicio:

Tu tarea es construir un modelo de clasificación usando el algoritmo **Random Forest** que, dado un conjunto de características de un proyecto tecnológico, prediga cuál es el **lenguaje de programación más adecuado**.

□ Parte 1: Preparar los datos

Los datos se representan como arrays de NumPy. Cada proyecto tiene estas características:

Atributo	Tipo	Descripción
velocidad_requerida	float (0-1)	¿Qué tan importante es la velocidad de ejecución?
facilidad_mantenimiento	float (0-1)	¿Qué tan fácil debe ser mantener el código?
disponibilidad_libs	float (0-1)	¿Qué tan importante es que haya muchas librerías?
tipo_app	int (0-2)	0: web, 1: escritorio, 2: sistema embebido
rendimiento_deseado	float (0-1)	¿Qué tan importante es el rendimiento general?

La **etiqueta** será un número entero que representa el lenguaje:

- 0 → Python
- 1 → JavaScript
- 2 → Java

- 3 → C++

🧠 Parte 2: Crear el modelo

1. Genera un conjunto de datos de ejemplo (puedes crear 100 proyectos artificiales con NumPy).
2. Entrena un modelo de **Random Forest** (`sklearn.ensemble.RandomForestClassifier`).
3. Implementa una función que reciba un array con las 5 características de un proyecto nuevo y devuelva el lenguaje más adecuado.

🎯 Objetivo final:

Implementa una clase llamada `LanguagePredictor` que tenga:

```
class LanguagePredictor:
    def __init__(self):
        ...

    def train(self, X, y):
        ...

    def predict(self, features: np.ndarray) -> str:
        ...
```

💡 Sugerencia para el alumno:

- Usa `numpy.random.rand` para generar las características (entre 0 y 1).
- Usa `numpy.random.randint` para crear el tipo de app (0, 1 o 2).
- Genera etiquetas de forma algo correlacionada: por ejemplo, si `rendimiento_deseado` > 0.8 y tipo_app == 2, tal vez el lenguaje sea C++.
- Usa un diccionario para traducir el resultado del modelo (número) al lenguaje de programación.

🧪 3. Ejemplo de uso

```
# Generar datos y entrenar
X, y = generate_dataset()
predictor = LanguagePredictor()
predictor.train(X, y)

# Crear un proyecto nuevo
new_project = np.array([0.7, 0.9, 0.5, 1, 0.6])  # Características del proyecto

```

```
# Predecir lenguaje ideal
pred = predictor.predict(new_project)
print(f"Lenguaje recomendado para el nuevo proyecto: {pred}")
```

 Salida esperada:

```
Lenguaje recomendado para el nuevo proyecto: JavaScript
```

37. Detectar jugadores con potencial profesional
□ Ejercicio: ¿Quién será un jugador profesional? - Clasificación con SVM

□ Contexto:

Imagina que trabajas en una plataforma de eSports y tu equipo está desarrollando un sistema de scouting para detectar **jugadores con potencial profesional** en base a sus estadísticas de juego.

Tu tarea es construir un **modelo de clasificación usando SVM (Support Vector Machine)** que, dada la información de un jugador, prediga si tiene el perfil de **jugador profesional** (`1`) o no (`0`).

□ Objetivo del ejercicio

Implementar un clasificador que:

- Use datos simulados de jugadores (partidas ganadas, horas jugadas, precisión, velocidad de reacción, estrategia).
- Entrene un modelo de SVM con `scikit-learn`.
- Clasifique a nuevos jugadores como **"pro"** o **"casual"**.
- Evalúe el modelo con métricas de precisión.

□ Datos de entrada:

Cada jugador se representa con las siguientes características (todas normalizadas entre 0 y 1):

Atributo	Descripción
`win_rate`	Porcentaje de partidas ganadas
`hours_played`	Total de horas jugadas
`accuracy`	Precisión al disparar
`reaction_time`	Velocidad de reacción (más bajo es mejor)
`strategic_play`	Capacidad de planificación estratégica

□ Ejemplo de datos

```
simulator = GameSimulator()
simulator.run()
```

Salida esperada

```
Jugador profesional:
Precisión del modelo: 1.00
```

 Tareas a realizar:

1. Implementa la clase `ProPlayerClassifier` con los métodos:
    - `train(X, y)` para entrenar el modelo.
    - `predict(player_stats)` para predecir si un jugador es profesional.
2. Usa `sklearn.svm.SVC` como modelo base.
3. Prueba el modelo con al menos 2 predicciones distintas.
4. Evalúa el rendimiento con `accuracy_score`.

38. Clasificador de calidad del aire
Contexto:

Imagina que trabajas en una empresa de tecnología verde. Te piden crear un modelo que, basándose en datos de calidad del aire, prediga si un área es **saludable** o **contaminada** para las personas.

Objetivo:

Usando **NumPy** para generar datos simulados y **SVM** para clasificar, debes construir un modelo que prediga si un área tiene **buena** o **mala** calidad del aire.

Requisitos:

- Crear una clase `AirSample` con los atributos:
  - `pm25` (partículas finas PM2.5 en µg/m³)
  - `pm10` (partículas gruesas PM10 en µg/m³)
  - `o3` (ozono en ppb)
  - `no2` (dióxido de nitrógeno en ppb)
  - `quality` (0 = saludable, 1 = contaminado) → solo en entrenamiento.
- Crear una clase `AirDataGenerator` para generar datos sintéticos:
  - Regla orientativa: si `pm25 > 35` o `pm10 > 50` o `no2 > 40` → contaminado.
- Crear una clase `AirQualityClassifier` para:
  - Entrenar un modelo SVM usando los datos generados.
  - Predecir la calidad del aire de una nueva muestra.
- Crear una clase `AirQualityExample` para:
  - Generar muestras.
  - Entrenar el clasificador.
  - Predecir la calidad de una muestra nueva (creada manualmente).
  - Mostrar el resultado de la predicción.

Ejemplo de uso

```
example = AirQualityExample()
example.run()
```

Salida esperada

```
 Muestra de aire:
PM2.5: 22, PM10: 30, O3: 50, NO2: 35
```

3. 🔍 Predicción de calidad: Saludable 🔍

39. Clasificar piezas industriales

🧠 **Contexto:**

Eres parte del equipo de visión por computador de una fábrica que produce piezas metálicas de precisión.

Cada pieza fabricada se analiza por una cámara de inspección que extrae 4 métricas clave:

- **Textura** (entre 0 y 1): mide la homogeneidad superficial.
- **Simetría** (entre 0 y 1): cuán simétrica es la pieza en sus formas.
- **Bordes detectados** (valor entero): cantidad de bordes identificados en la imagen (representa rugosidad o detalles).
- **Desviación del centro** (`offset`): qué tan descentrada está la pieza respecto al ideal (valor positivo, suele ir de 0 a 0.5).

Con estos datos, tu tarea es **entrenar un modelo de clasificación basado en Máquinas de Vectores de Soporte (SVM)** para identificar automáticamente si una pieza es **"Correcta"** o **"Defectuosa"**.

📏 **Reglas de clasificación (para generar los datos):**

Al generar tus datos, considera lo siguiente:

- Si una pieza tiene **simetría < 0.4** *y* un **offset > 0.25**, es **defectuosa**.
- Si tiene **textura < 0.35**, o **bordes < 30**, o **offset > 0.35**, también es **defectuosa**.
- El resto de piezas se consideran **correctas**.

🛠️ **Tu objetivo:**

1. Simula un dataset de 400 piezas usando `NumPy`, con los siguientes parámetros:
   - Textura: media `0.5`, desviación `0.15`
   - Simetría: media `0.6`, desviación `0.2`
   - Bordes: media `50`, desviación `15`
   - Offset: media `0`, desviación `0.2` (usa `np.abs()` para obtener valores positivos)
2. Crea una clase `Piece` que almacene los valores y convierta una pieza a un vector de entrada.
3. Implementa una clase `PieceDatasetGenerator` que genere automáticamente los datos y etiquetas según las reglas.
4. Crea una clase `PieceClassifier` que:
   - Entrene un modelo `SVC` con kernel `'rbf'`

- Permita hacer predicciones sobre nuevas piezas
- Evalúe el modelo con `confusion_matrix` y `classification_report`

5. Divide el dataset en entrenamiento (70%) y prueba (30%) con `train_test_split`.
6. Prueba el modelo con una pieza personalizada:
    a.
    1. `textura = 0.45`
    2. `simetría = 0.5`
    3. `bordes = 45`
    4. `offset = 0.15`
7. Visualiza los resultados con `matplotlib`, graficando las piezas según:
    - Eje X: textura
    - Eje Y: offset
    - Color según su etiqueta (“Correcta” o “Defectuosa”)

□ Ejemplo de uso

```
example = PieceAnalysisExample()
example.run()
```

Salida esperada

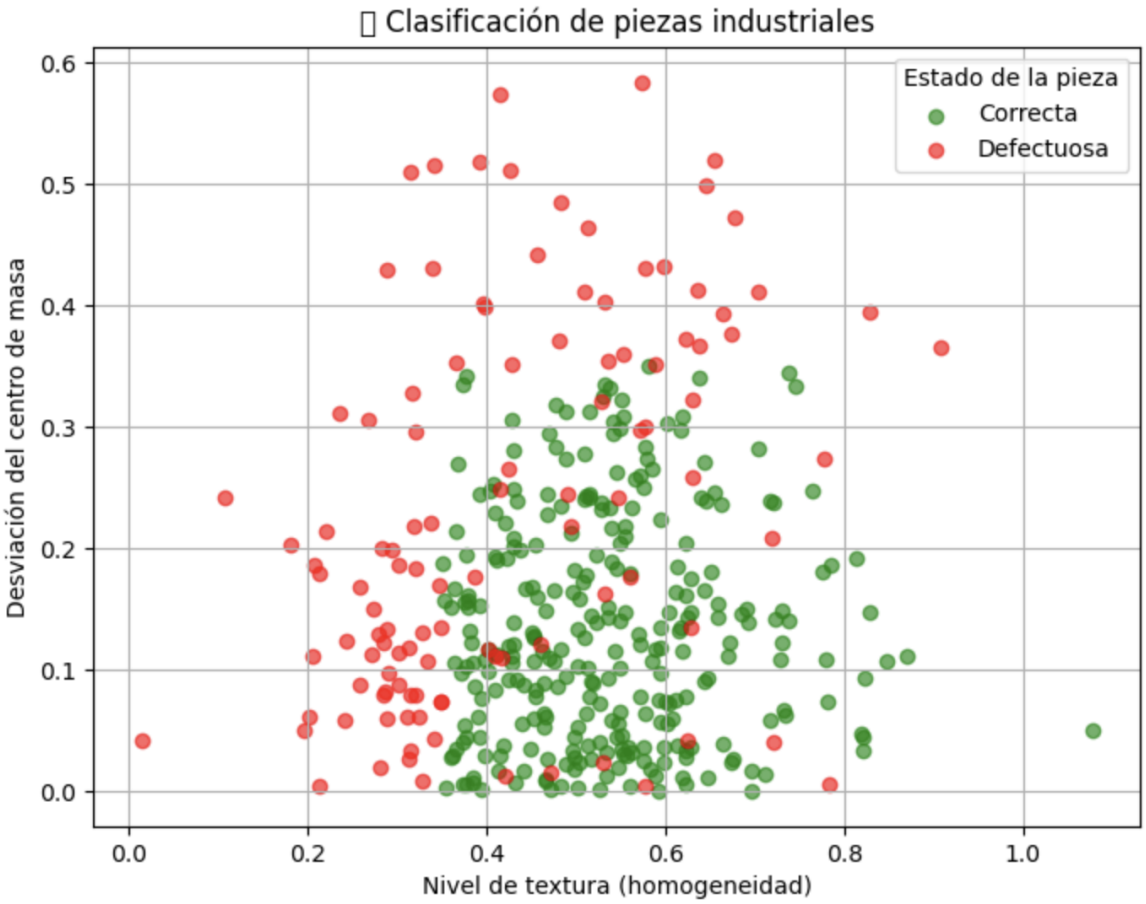
```
□ Matriz de confusión:
[[87  0]
 [28  5]]

□ Informe de clasificación:
              precision    recall  f1-score   support

     Correcta       0.76      1.00      0.86        87
   Defectuosa       1.00      0.15      0.26        33

    accuracy                           0.77       120
   macro avg       0.88      0.58      0.56       120
weighted avg       0.82      0.77      0.70       120


□ Predicción de pieza personalizada:
  → Textura: 0.45, Simetría: 0.50, Bordes: 45, Offset: 0.15
  → Clasificación: Correcta
```


Clasificación de piezas industriales
Estado de la pieza
Correcta
Defectuosa
Desviación del centro de masa
Nivel de textura (homogeneidad)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.0
0.2
0.4
0.6
0.8
1.0

K-MEDIAS

40. Algoritmo de k-medias
El objetivo es implementar una función que:

1. **Entrene un modelo K-Means** (`KMeans` de `sklearn.cluster`).
2. **Agrupe los datos en `k` clusters**.
3. **Evalúe el rendimiento** del modelo utilizando:
    - **Inercia** (`inertia_`): mide qué tan bien se agrupan los datos.
    - **Puntuación de Silueta** (`silhouette_score`): evalúa qué tan bien están separados los clusters.
    - **Precisión ajustada (`adjusted_rand_score`)**: mide la similitud con las etiquetas reales.
4. **Devuelva los resultados en un diccionario**.
5. **Supervise la implementación con pruebas unitarias (`unittest`)**.

Instrucciones

1. Implementa una función llamada `entrenar_y_evaluar_kmeans(X, y, k)` que:
    - Entrene un modelo `KMeans(n_clusters=k, random_state=42)`.
    - Asigne cada muestra a un cluster (`labels_`).
    - Calcule las métricas de evaluación mencionadas.
    - Devuelva un diccionario con:
        - `"clusters"`: Array con las asignaciones de cluster para cada muestra.
        - `"inertia"`: Suma de las distancias al centroide más cercano.
            - `inertia = modelo.inertia_`
        - `"silhouette_score"`: Puntuación de silueta (qué tan bien se separan los clusters).
            - `silhouette = silhouette_score(X, clusters)`
        - `"adjusted_rand_score"`: Similaridad con las etiquetas reales.
            - `rand_score = adjusted_rand_score(y, clusters)`
2. **Usa el dataset de flores `iris`** de `sklearn.datasets`.
3. **Asegúrate de que `k=3` (correspondiente a las 3 clases reales de Iris)**.

41. Agrupando jugadores
¿Qué tipo de gamer eres?"

□ Enunciado del ejercicio

Eres parte del equipo de análisis de una plataforma de videojuegos que quiere entender mejor a sus usuarios. Se ha recopilado información sobre distintos jugadores basada en su comportamiento dentro del juego. Tu misión es **agrupar a estos jugadores en diferentes tipos (clusters)** según su estilo de juego, utilizando el algoritmo de **K-Means**.

□ Tareas a realizar

1. **Crea una clase `Player` que contenga los siguientes atributos:**
   - `name` (str): nombre del jugador
   - `avg_session_time` (float): tiempo medio de juego por sesión (en horas)
   - `missions_completed` (int): número de misiones completadas
   - `accuracy` (float): precisión de disparo (entre 0 y 1)
   - `aggressiveness` (float): valor entre 0 (pasivo) y 1 (muy agresivo)
2. **Crea una clase `PlayerClusterer` con los siguientes métodos:**
   - `fit(players: List[Player], n_clusters: int)`: entrena un modelo K-Means con los datos de los jugadores.
   - `predict(player: Player) -> int`: devuelve el número de cluster al que pertenece un nuevo jugador.
   - `get_cluster_centers()`: devuelve los centros de los clusters.
   - `print_cluster_summary(players: List[Player])`: imprime qué jugadores hay en cada grupo.
3. **Usa los datos proporcionados a continuación para entrenar el modelo con 3 clusters:**

```
data = [
            ("Alice", 2.5, 100, 0.85, 0.3),
            ("Bob", 1.0, 20, 0.60, 0.7),
            ("Charlie", 3.0, 150, 0.9, 0.2),
            ("Diana", 0.8, 15, 0.55, 0.9),
            ("Eve", 2.7, 120, 0.88, 0.25),
            ("Frank", 1.1, 30, 0.62, 0.65),
            ("Grace", 0.9, 18, 0.58, 0.85),
            ("Hank", 3.2, 160, 0.91, 0.15)
        ]
```

4. **Crea una clase `GameAnalytics` que haga lo siguiente:**
   - Cree los objetos `Player` con los datos anteriores.
   - Cree un objeto `PlayerClusterer`, entrene el modelo y muestre los clusters formados.
   - Prediga el cluster para un nuevo jugador: `("Zoe", 1.5, 45, 0.65, 0.5)`.

□ Requisitos del ejercicio

- Utiliza `scikit-learn` (`KMeans`) para la agrupación.
- Usa programación orientada a objetos.

- No uses ficheros externos. Todo debe estar en el código.
- Asegúrate de imprimir resultados entendibles para los usuarios.

Ejemplo de uso

```
analytics = GameAnalytics()
analytics.run()
```

Salida esperada

```
Cluster 2:
  - Alice
  - Eve
Cluster 1:
  - Bob
  - Diana
  - Frank
  - Grace
Cluster 0:
  - Charlie
  - Hank

Jugador Zoe pertenece al cluster: 1
```

42. Agrupar viajeros según sus preferencias
Imagina que trabajas en una agencia de viajes internacional que recibe cientos de perfiles de viajeros. Cada viajero indica cuánto le gustan diferentes tipos de destinos: **playa**, **montaña**, **ciudad** y **campo**.

Tu misión es:

 Crear una **clase** `Traveler` para representar un viajero, con atributos como:

- `beach`
- `mountain`
- `city`
- `countryside`

 Crear una **clase** `TravelerGenerator` que pueda **generar automáticamente** viajeros con gustos aleatorios (usando `numpy`).

 Crear una **clase** `TravelerClusterer` que use **K-Means Clustering** (`sklearn`) para **agrupar** a los viajeros en **3 grupos principales** según sus preferencias.

 Permitir probar con un viajero nuevo: dado un viajero personalizado, deberás predecir **a qué grupo** pertenecería.

 Crear una **clase** `TravelerClusteringExample` que **integre todo**:

- Genere los datos de entrenamiento.
- Entrene el modelo de agrupamiento.
- Cree un viajero nuevo.
- Prediga a qué grupo pertenece ese viajero y **muestre los resultados**.

 Requisitos mínimos:

- `Traveler`: clase que almacena las preferencias.
- `TravelerGenerator`: clase que genera viajeros de manera aleatoria.
- `TravelerClusterer`: clase que entrena el modelo K-Means y predice el grupo de nuevos viajeros.
- `TravelerClusteringExample`: clase que coordina todo el flujo y ejecuta el ejemplo completo.

 Pistas para completar la misión:

- Usa `numpy.random.uniform(0, 10)` para generar los gustos aleatorios.
- Usa `sklearn.cluster.KMeans(n_clusters=3)` para entrenar el modelo.
- Representa cada viajero como un **vector de 4 números** (uno por tipo de preferencia).

- Al final del ejemplo, imprime en pantalla el grupo asignado al nuevo viajero.

□ Ejemplo de uso

```
# Ejecutar ejemplo
example = TravelerClusteringExample()
example.run()
```

Salida esperada

```
□□□□ Cluster Centers (Preferencias promedio):
Cluster 0: Playa=4.79, Montaña=5.16, Ciudad=7.79, Campo=7.82
Cluster 1: Playa=5.11, Montaña=5.54, Ciudad=6.60, Campo=1.66
Cluster 2: Playa=4.69, Montaña=5.23, Ciudad=1.46, Campo=6.16

Interpretación aproximada:
- Cluster con alta Playa y Ciudad: Viajero urbano y costero.
- Cluster con alta Montaña y Campo: Amante de la naturaleza.
- Cluster equilibrado: Viajero versátil o aventurero.

□ Nuevo viajero con preferencias:
Playa: 9, Montaña: 2, Ciudad: 8, Campo: 1
□ El nuevo viajero pertenece al grupo 1.
```

43. Agrupar perfiles de sueño
□ Objetivo del ejercicio:

Analizar perfiles de sueño simulados de distintas personas para agruparlos según sus patrones de descanso utilizando el algoritmo de **k-medias (K-Means)**.

El objetivo es encontrar **tipos de durmientes** (como los que duermen mucho, los que se despiertan frecuentemente, etc.) a partir de sus datos personales.

□ Contexto:

Un equipo de investigadores del sueño quiere estudiar cómo duermen diferentes personas durante varias noches.

Para ello, han registrado características clave como:

- **Duración del sueño** (en horas)
- **Latencia**: cuántos minutos tarda la persona en dormirse
- **Cantidad de despertares por noche**
- **Variabilidad**: cuánto varía la hora a la que se acuesta cada día

Tu tarea es **agrupar estos perfiles en 3 tipos de durmientes** usando K-Means, y visualizar los resultados de manera clara y útil.

□ Datos simulados:

Los datos serán generados aleatoriamente con estas distribuciones:

Variable	Descripción	Distribución
Duración	Horas de sueño por noche	Normal(7, 1.2)
Latencia	Minutos en quedarse dormido	Normal(20, 10), abs()
Despertares	Número de veces que se despierta	Poisson(1.5)
Variabilidad	Variación diaria de hora de acostarse	Normal(30, 15), abs()

□ Estructura sugerida del código:

Clases utilizadas:

1. `SleepProfile`
   Representa un perfil individual de sueño, con atributos como duración del sueño, latencia, despertares y variabilidad.
   - Método `to_vector()` convierte estos atributos en un vector numérico para el algoritmo.
2. `SleepDatasetGenerator`
   Genera `n` perfiles de sueño aleatorios con características estadísticas realistas.
   - Usa distribución normal y de Poisson para crear variedad en los datos.
3. `SleepClusterer`
   Se encarga de escalar los datos y aplicar el algoritmo K-Means para agrupar los perfiles en `k` grupos.
   - Usa `StandardScaler` para normalizar.
   - Devuelve las etiquetas de grupo y los datos escalados.
4. `SleepAnalysisExample`
   Es la clase que integra todo: genera los datos, entrena el modelo, agrupa y visualiza los resultados en un gráfico.
   - También imprime los centroides de cada grupo interpretados en la escala original.

□ ¿Qué deberías mostrar al final?

1. Los **centroides de los grupos**, interpretando lo que caracteriza a cada uno.
2. Un **gráfico de dispersión** donde se vea la agrupación de perfiles por:
   - Eje X: Duración del sueño
   - Eje Y: Variabilidad de hora de dormir

3. Comentarios sobre posibles tipos de durmientes: ¿hay un grupo de "insomnes"? ¿otro de "buenos durmientes"?

□ Ejemplo de uso

```
example = SleepAnalysisExample()
example.run()
```

Salida esperada

```
□ Centroides de los grupos:
Grupo 0: Duración=6.30h, Latencia=19.3min, Despertares=1.2, Variabilidad=39.6min
Grupo 1: Duración=6.79h, Latencia=18.9min, Despertares=3.4, Variabilidad=26.5min
Grupo 2: Duración=7.98h, Latencia=18.6min, Despertares=1.0, Variabilidad=22.3min
```

EJERCICIOS FINALES

44. Predicción de Abandono Escolar

La universidad ha recopilado datos de estudiantes para intentar predecir si un alumno abandonará sus estudios o no. Se te proporciona un conjunto de datos generado dentro del código.

Tareas a realizar:

1. **Crear una función** `entrenar_modelo(data)` que:
   - Reciba un DataFrame con los datos de los estudiantes.
   - Entrene un modelo de Machine Learning adecuado.
   - Evalúe su precisión con métricas adecuadas.
   - Devuelva el modelo entrenado.
2. **Probar la función** y analizar los resultados.
3. **Hacer una predicción** con un nuevo estudiante.

La salida propuestas debe ser similar a esta:

```
Precisión del modelo: 0.57
Reporte de clasificación:
              precision    recall  f1-score   support

           0       0.59      0.96      0.73        24
           1       0.00      0.00      0.00        16
```

```
7.
8.      accuracy                           0.57        40
9.     macro avg       0.29      0.48      0.37        40
10. weighted avg       0.35      0.57      0.44        40
11.
12. El estudiante probablemente: Seguirá estudiando
```

45. Predicción del Precio de una Vivienda
Enunciado

Un banco tiene un conjunto de datos con información sobre viviendas y quiere predecir el **precio** de una casa basada en ciertas características. Las características disponibles son las siguientes:

- **Superficie**: Área de la vivienda en metros cuadrados.
- **Número de habitaciones**: Cantidad de habitaciones en la casa.
- **Antigüedad**: Número de años desde que la vivienda fue construida.
- **Distancia al centro de la ciudad**: Distancia en kilómetros a la plaza principal de la ciudad.
- **Número de baños**: Cantidad de baños en la vivienda.
- **Precio**: Precio de la vivienda (variable objetivo).

Objetivo del Ejercicio

1. **Entrenar un modelo de regresión** para predecir el precio de las viviendas en función de las características proporcionadas.
2. **Evaluar el modelo** utilizando métricas adecuadas para regresión, como el **Error Cuadrático Medio (MSE)** y el $\mathbf{R^2}$.
3. **Realizar predicciones** con nuevas viviendas para estimar su precio según las características de la vivienda.

Datos Proporcionados (Simulados)

Los datos contienen 200 ejemplos de viviendas con las características mencionadas. Los datos están disponibles en el código, y los estudiantes deben decidir qué preprocesamiento aplicar y qué modelo de predicción utilizar.

La solución aproximada es

```
Error Cuadrático Medio (MSE): 14748907009.71
R² del modelo: 0.02
El precio estimado de la vivienda es: $284,716.76
```

46. Agrupación de clientes según comportamientos de compra
Contexto:

Trabajas en una empresa de análisis de datos para una tienda en línea. La empresa quiere mejorar su estrategia de marketing personalizando las ofertas y promociones para diferentes tipos de clientes, basándose en su comportamiento de compra. El objetivo es identificar grupos de clientes que tengan patrones de compra similares, lo que permitirá crear campañas de marketing más efectivas y dirigidas.

Objetivo:

Tu tarea es desarrollar un sistema que agrupe a los clientes según sus características de compra. Las características de cada cliente están relacionadas con su comportamiento a lo largo del último año, y son las siguientes:

- **Monto total gastado en el último año** (en dólares).
- **Frecuencia de compras** (número de compras realizadas en el último año).
- **Categorías de productos preferidas** (número de compras realizadas en diferentes categorías como: tecnología, ropa, alimentos, entre otras).

Con estas características, tu objetivo es construir un modelo que agrupe a los clientes en **clusters** (grupos) que compartan comportamientos similares.

Instrucciones:

Deberás implementar tres funciones principales para lograr el objetivo:

1. `generar_datos_clientes(num_muestras)`:
    - Genera un conjunto de datos ficticio con `num_muestras` clientes.
    - Cada cliente debe tener las siguientes características:
        - **Monto total gastado**: Un valor numérico entre 100 y 10,000 dólares.
        - **Frecuencia de compras**: Un valor entre 1 y 100, que representa el número de compras realizadas durante el último año.
        - **Categorías preferidas**: Un conjunto de números que represente la cantidad de compras realizadas en diferentes categorías de productos (por ejemplo: tecnología, ropa, alimentos, etc.). Puedes generar entre 3 y 5 categorías.
2. `entrenar_modelo_cluster(data)`:
    - Esta función recibe los datos generados por la función `generar_datos_clientes`.
    - Utiliza un algoritmo de **agrupamiento no supervisado** para descubrir patrones en los datos y dividir a los clientes en diferentes **clusters**. El modelo debe ser capaz de agrupar a los clientes en grupos significativos según sus comportamientos de compra.
    - **Nota**: El número de clusters no está predeterminado, por lo que tendrás que elegir un enfoque que permita decidir cuántos clusters es adecuado para los datos.
3. `predecir_cluster(modelo, cliente)`:
    - Esta función recibe un cliente nuevo (con sus características de monto gastado, frecuencia de compras y categorías preferidas) y el modelo entrenado.
    - La función debe predecir a qué cluster pertenece este cliente, es decir, debe identificar el grupo al que este cliente tiene más similitudes con los demás clientes del conjunto entrenado.

Consideraciones:

- **Elección del número de clusters**: El número de clusters no está especificado y es una decisión que deberás tomar durante el desarrollo del modelo. Puedes explorar diferentes técnicas, como el **método del codo**, para determinar el número óptimo de clusters.
- **Evaluación del modelo**: Aunque este es un modelo no supervisado, puedes evaluar la calidad de los grupos observando qué tan homogéneos son los clientes dentro de cada cluster y qué tan diferentes son los clusters entre sí.

47. Predicción para acertar la lotería
Objetivo:

En este ejercicio, se te solicita aplicar técnicas de Machine Learning para predecir cuál de varias combinaciones de números tiene mayor probabilidad de éxito en un escenario simulado de lotería. Deberás implementar varias funciones y entrenar un modelo para realizar la predicción.

Requisitos:

Tu tarea es implementar **dos funciones principales** para resolver este ejercicio:

1. **Función** `generar_series(num_series)`:
    - Esta función debe generar una cantidad de combinaciones de 6 números aleatorios entre 1 y 49.
    - Cada combinación de números representará una serie de lotería.
2. **Función** `entrenar_modelo()`:
    - Esta función debe simular un conjunto de datos de entrenamiento con combinaciones de lotería y un resultado de éxito o fracaso (10% de éxito, 90% de fracaso).
    - Debes entrenar un modelo de clasificación utilizando los datos simulados.
    - La función debe devolver un modelo entrenado, que podrás utilizar para predecir la probabilidad de éxito de nuevas combinaciones.
3. **Función** `predecir_mejor_serie(modelo, num_series)`:
    - Usando el modelo entrenado, esta función debe predecir cuál de las combinaciones generadas tiene mayor probabilidad de éxito.
    - La función debe devolver la combinación de números con mayor probabilidad de éxito, así como esa probabilidad.

Datos Simulados:

- Se generarán **1000 combinaciones de números de lotería** con 6 números aleatorios cada una.
- Para cada combinación, se asignará una etiqueta de **éxito (1)** o **fracaso (0)** de acuerdo con una probabilidad del **10% de éxito** y un **90% de fracaso**.
- El objetivo es predecir cuál de las combinaciones generadas tendrá mayor probabilidad de éxito en un escenario hipotético.

Pasos a seguir:

1. **Generar combinaciones**: La función `generar_series(num_series)` debe crear una lista de combinaciones de 6 números, donde `num_series` es el número de combinaciones a generar.
2. **Entrenar el modelo**: La función `entrenar_modelo()` debe entrenar un modelo de Machine Learning para clasificar las combinaciones como de éxito o fracaso. El modelo debe predecir el éxito de nuevas combinaciones.
3. **Predecir la mejor serie**: La función `predecir_mejor_serie(modelo, num_series)` debe generar `num_series` combinaciones y usar el modelo para predecir cuál tiene la mayor probabilidad de éxito. Debe devolver la mejor serie con su probabilidad de éxito.

48. Clasificación Automática de Frutas
Ejercicio: Clasificación Automática de Frutas □□□

Contexto:

Trabajas en una empresa de distribución de frutas y necesitas desarrollar un sistema que pueda clasificar automáticamente distintos tipos de frutas en función de sus características físicas. Para ello, deberás analizar un conjunto de datos con información sobre frutas y diseñar un modelo de predicción adecuado.

Objetivo:

Tu tarea es desarrollar un sistema de clasificación que, dado el peso y el tamaño de una fruta, pueda predecir de qué tipo se trata. Para ello, deberás implementar una o varias funciones en Python que permitan:

1. **Generar un conjunto de datos de frutas** con características realistas.
2. **Entrenar un modelo de Machine Learning** para clasificar las frutas en diferentes categorías.
3. **Hacer predicciones** con nuevas frutas basadas en sus características.

Instrucciones:

1. **Crea la función `generar_datos_frutas(num_muestras)`:**
   - Debe generar un conjunto de datos ficticio con `num_muestras` frutas.
   - Cada fruta tendrá un peso (en gramos) y un tamaño (en cm).
   - Se le asignará una etiqueta según el tipo de fruta (por ejemplo, Manzana, Plátano, Naranja).
2. **Desarrolla la función `entrenar_modelo(data)`:**

    - Recibe el conjunto de datos y entrena un modelo de Machine Learning para clasificar frutas según su peso y tamaño.
    - Elige el enfoque que consideres más adecuado para resolver el problema.
3. **Implementa la función `predecir_fruta(modelo, peso, tamaño)`:**
    - Usa el modelo entrenado para predecir a qué tipo de fruta pertenece un nuevo ejemplo dado su peso y tamaño.

Consideraciones:

- No se te indica qué algoritmo utilizar, por lo que debes analizar el problema y elegir la mejor estrategia para resolverlo.
- Puedes probar diferentes enfoques y comparar su precisión.

 Preguntas para reflexionar:

- ¿Cómo determinaste qué modelo de Machine Learning usar?
- ¿Cómo podrías mejorar la precisión del sistema?
- ¿Qué otras características podrían influir en la clasificación de frutas?

 Pista: Puedes explorar modelos supervisados como clasificación o métodos basados en distancias.

49. Predicción de compra de un producto en línea
Ejercicio: Predicción de Compra de un Producto en Línea

Contexto:

Eres parte del equipo de análisis de datos en una tienda en línea. Tu objetivo es predecir si un usuario hará una compra o no basándote en su comportamiento en el sitio web. La tienda recopila datos sobre el comportamiento de los usuarios en el sitio, como el número de páginas que visitan y el tiempo que pasan en la página. Te piden que desarrolles un sistema que utilice estos datos para predecir la probabilidad de que un usuario compre un producto.

Objetivo:

Tu tarea es desarrollar un sistema que, a partir de la información sobre el comportamiento de los usuarios, prediga si comprarán o no un producto. Debes crear varias funciones que permitan:

1. **Generar un conjunto de datos sintéticos** sobre el comportamiento de los usuarios en el sitio.
2. **Entrenar un modelo de predicción** basado en el comportamiento de los usuarios.
3. **Realizar predicciones** sobre si un nuevo usuario comprará o no el producto en función de su actividad en el sitio web.

Funciones que debes implementar:

1. **Función `generar_datos_compras(num_muestras)`**:
    - **Descripción**: Esta función debe generar un conjunto de datos sintéticos que representen el comportamiento de los usuarios en el sitio web.

- **Entrada**: Un parámetro `num_muestras` que indica el número de registros de usuarios que quieres generar.
- **Proceso**:
  - Cada usuario tendrá dos características:
    - `num_paginas_vistas`: Un valor entero entre 1 y 20, que representa el número de páginas que un usuario ha visitado en el sitio.
    - `tiempo_en_sitio`: Un valor decimal entre 0 y 30 minutos, que representa el tiempo total que el usuario pasó en el sitio.
  - La variable objetivo (`etiqueta`) se asigna como:
    - **1** si el número de páginas vistas es mayor a 5 y el tiempo en el sitio es mayor a 10 minutos.
    - **0** si el número de páginas vistas es 5 o menos o si el tiempo en el sitio es 10 minutos o menos.
- **Salida**: Un par de arrays:
  - Un array con las características de cada usuario (número de páginas vistas y tiempo en el sitio).
  - Un array con las etiquetas de compra (1 o 0).

2. **Función** `entrenar_modelo(datos)`:
   - **Descripción**: Esta función debe entrenar un modelo de Machine Learning para predecir la compra del usuario basado en las características generadas.
   - **Entrada**: El conjunto de datos generado por la función anterior. Debe contener las características (`num_paginas_vistas`, `tiempo_en_sitio`) y las etiquetas (`0` o `1`).
   - **Proceso**:
     - Divide el conjunto de datos en dos partes: un conjunto de entrenamiento y un conjunto de prueba (usualmente 70% entrenamiento, 30% prueba).
     - Utiliza un algoritmo de clasificación (como la regresión logística) para entrenar el modelo.
     - Entrena el modelo utilizando las características (`num_paginas_vistas` y `tiempo_en_sitio`) y las etiquetas (`compra` o `no compra`).
   - **Salida**: El modelo entrenado.
3. **Función** `predecir_compra(modelo, num_paginas_vistas, tiempo_en_sitio)`:
   - **Descripción**: Esta función debe predecir si un usuario comprará o no el producto basándose en sus características (número de páginas vistas y tiempo en el sitio).
   - **Entrada**: El modelo entrenado y las características de un nuevo usuario:
     - `num_paginas_vistas`: El número de páginas que un nuevo usuario ha visitado en el sitio.
     - `tiempo_en_sitio`: El tiempo que el nuevo usuario ha pasado en el sitio, en minutos.
   - **Salida**: Devuelve la predicción de la compra, que puede ser **1** (comprará) o **0** (no comprará).
4. **Función** `evaluar_modelo(modelo, datos)`:
   - **Descripción**: Esta función debe evaluar el rendimiento del modelo entrenado.
   - **Entrada**: El modelo entrenado y el conjunto de datos original (conjunto de características y etiquetas).
   - **Proceso**:

        - Divide el conjunto de datos en un conjunto de entrenamiento y uno de prueba.
        - Realiza las predicciones sobre el conjunto de prueba.
        - Calcula y muestra la **precisión** del modelo, es decir, la proporción de predicciones correctas (puedes usar la métrica de precisión o exactitud).
    - **Salida**: Muestra la precisión del modelo y devuelve el valor numérico de precisión.

Ejemplo de Flujo:

1. Genera un conjunto de datos de usuarios usando `generar_datos_compras(100)`.
2. Entrena el modelo con el conjunto de datos generado usando `entrenar_modelo(datos)`.
3. Evalúa la precisión del modelo usando `evaluar_modelo(modelo, datos)`.
4. Realiza predicciones sobre nuevos usuarios usando `predecir_compra(modelo, 8, 12)` para predecir si un usuario que vio 8 páginas y pasó 12 minutos en el sitio hará una compra.

50. Detectar correo electrónico spam
Clasificación de Emails: ¿Spam o No Spam?

Contexto: Tienes un conjunto de datos que contiene información sobre emails. Cada email tiene un conjunto de características, como la longitud del mensaje, la frecuencia de ciertas palabras clave, la cantidad de enlaces, y otros aspectos relevantes. El objetivo es construir un modelo de clasificación para predecir si un email es **Spam** o **No Spam**.

Objetivo: Tu tarea es implementar un modelo de clasificación que, dada la información de un email (características como la longitud del mensaje y la frecuencia de palabras clave), sea capaz de predecir si el email es **Spam** (1) o **No Spam** (0).

Funciones a Implementar:

1. **Generar datos de emails:**

   Función: `generar_datos_emails(num_muestras)`

    - Esta función debe generar un conjunto de datos ficticios con **num_muestras** emails.
    - Cada email tendrá las siguientes características:
        - **longitud_mensaje**: Un número aleatorio que representa la longitud del email en caracteres (entre 50 y 500).
        - **frecuencia_palabra_clave**: Un número aleatorio que representa la frecuencia de una palabra clave relacionada con spam (entre 0 y 1).
        - **cantidad_enlaces**: Un número aleatorio que representa la cantidad de enlaces en el email (entre 0 y 10).
    - Cada email será etiquetado como **Spam (1)** o **No Spam (0)**.
2. **Entrenar el modelo SVM:**

Función: `entrenar_modelo_svm(datos, etiquetas)`

- Esta función debe tomar un conjunto de datos con características de emails y sus etiquetas, y entrenar un modelo de clasificación.
- La salida debe ser el modelo entrenado.

3. **Realizar predicciones:**

   Función: `predecir_email(modelo, longitud_mensaje, frecuencia_palabra_clave, cantidad_enlaces)`

   - Esta función debe tomar un modelo entrenado y las características de un nuevo email, y devolver si el email es **Spam** o **No Spam**.
   - La salida debe ser una cadena de texto que indique si el email es **Spam** o **No Spam**.

Instrucciones:

1. **Generar Datos**: Para empezar, debes generar un conjunto de datos con emails etiquetados (Spam o No Spam).
2. **Entrenar el Modelo**: Entrenar el modelo de clasificación basado en las características del email.
3. **Predicciones**: Utiliza el modelo entrenado para predecir si un email es Spam o No Spam según sus características.

51. Segmentación de Clientes y Predicción de Compra
Segmentación de Clientes y Predicción de Comportamiento de Compra

Contexto

Una plataforma de comercio electrónico en crecimiento, quiere mejorar su estrategia de marketing mediante el análisis del comportamiento de sus clientes. El objetivo es segmentar a los clientes en grupos con características similares y predecir si un cliente realizará una compra en el próximo mes.

Para ello, utilizarás dos algoritmos de Machine Learning:

1. **K-Means** para agrupar a los clientes en segmentos basados en su historial de compras.
2. **Regresión Logística** para predecir si un cliente comprará en el próximo mes.

Objetivo del Proyecto

El objetivo es desarrollar un **modelo de segmentación y predicción de clientes** que permita a la empresa tomar mejores decisiones de marketing, ofreciendo promociones personalizadas a cada segmento de clientes.

Tareas principales:

1. **Generar un conjunto de datos sintético** con información de clientes.
2. **Implementar un modelo de segmentación** con K-Means para agrupar clientes.
3. **Entrenar un modelo de clasificación** con Regresión Logística para predecir compras futuras.
4. **Evaluar la precisión del modelo** y visualizar la matriz de confusión.
5. **Escribir pruebas unitarias** para verificar el correcto funcionamiento del modelo.

Datos Disponibles

Debes generar un conjunto de datos ficticio con **500 clientes**, donde cada cliente tiene las siguientes características:

- `total_spent` (*float*): Monto total gastado en la tienda (entre **$100 y $5000**).
- `total_purchases` (*int*): Número total de compras realizadas (**entre 1 y 100**).
- `purchase_frequency` (*float*): Promedio de días entre compras (**entre 1 y 30 días**).
- `will_buy_next_month` (*int*): Variable binaria (**1 = comprará, 0 = no comprará**), con una probabilidad del **30% de comprar y 70% de no comprar**.

Fases del Desarrollo

1. Generación de Datos

Crea un **conjunto de datos sintético** que cumpla con las características anteriores. Para esto, usa `numpy` y `pandas` para generar valores aleatorios realistas.

2. Implementar el Modelo de Segmentación (K-Means)

Usa el algoritmo **K-Means** para agrupar a los clientes en **tres segmentos** en función de:

- `total_spent` (gasto total).
- `total_purchases` (número de compras).
- `purchase_frequency` (frecuencia de compra).

Después de aplicar K-Means, agrega una nueva columna llamada `customer_segment` que indique el segmento asignado a cada cliente.

3. Implementar el Modelo de Predicción (Regresión Logística)

Después de la segmentación, usa **Regresión Logística** para predecir si un cliente realizará una compra en el próximo mes.

- Usa como variables de entrada (`X`) las características originales más la columna de segmentación.
- Usa `will_buy_next_month` como variable objetivo (`y`).
- Divide los datos en **conjunto de entrenamiento y prueba** (80%-20%).
- Entrena el modelo de Regresión Logística y evalúa su desempeño.

4. Evaluación del Modelo

Después del entrenamiento, mide el desempeño del modelo usando:

- **Precisión** (`accuracy_score`).
- **Matriz de confusión** (`confusion_matrix`).

La precisión del modelo debe ser al menos **60%** para considerar que tiene buen

rendimiento.

52. Diseñar una IA que entienda a los jugadores
Título: "Phantom Arena: Entrenando una IA para clasificar, predecir y agrupar jugadores"

Descripción:

En este ejercicio, deberás entrenar un modelo de Machine Learning para predecir el estilo de juego de un jugador, el número de victorias esperadas y asignar a cada jugador un grupo de características. Para ello, deberás crear una clase llamada `GameModel` que gestione y entrene los modelos correspondientes.

En lugar de cargar datos desde un archivo CSV, recibirás un conjunto de datos de prueba directamente en el código. Usarás estos datos para entrenar y evaluar tus modelos.

Tareas:

1. **Crear la clase `Player`:** Esta clase representará a un jugador y debe contener los siguientes atributos:
    - `player_name`: nombre del jugador (string).
    - `character_type`: tipo de personaje (string). Puede ser "mage", "tank", "archer", "assassin".
    - `avg_session_time`: tiempo promedio por sesión en minutos (float).
    - `matches_played`: número total de partidas jugadas (int).
    - `aggressive_actions`: cantidad de acciones agresivas realizadas (int).
    - `defensive_actions`: cantidad de acciones defensivas realizadas (int).
    - `items_bought`: cantidad de objetos comprados (int).
    - `victories`: número de victorias (int).
    - `style`: estilo de juego del jugador ("aggressive" o "strategic", sólo uno de estos).
2. **Crear la clase `GameModel`:** Esta clase debe ser capaz de:
    - Recibir una lista de jugadores y almacenarlos.

- Entrenar tres modelos diferentes:
  - **Modelo de clasificación**: para predecir el estilo de juego del jugador (`aggressive` o `strategic`).
  - **Modelo de regresión**: para predecir el número de victorias de un jugador.
  - **Modelo de clustering**: para asignar a cada jugador un grupo basado en sus características.
- Proveer métodos para:
  - Predecir el estilo de juego de un jugador.
  - Predecir las victorias de un jugador.
  - Asignar un cluster a un jugador.

3. **Pruebas:**
   - No deberás usar ningún archivo externo. Todos los datos serán proporcionados directamente en el código, en forma de una lista de objetos `Player`.
   - Después de entrenar los modelos, deberás hacer predicciones para un jugador de prueba.

Datos de prueba

Se te proporcionará un conjunto de datos de prueba como el siguiente:

```
players_data = [
    Player("P1", "mage", 40, 30, 90, 50, 20, 18, "aggressive"),
    Player("P2", "tank", 60, 45, 50, 120, 25, 24, "strategic"),
    Player("P3", "archer", 50, 35, 95, 60, 22, 20, "aggressive"),
    Player("P4", "tank", 55, 40, 60, 100, 28, 22, "strategic"),
]
```

Ejemplo de uso

```
# Crear datos de prueba para varios jugadores
players_data = [
    Player("P1", "mage", 40, 30, 90, 50, 20, 18, "aggressive"),
    Player("P2", "tank", 60, 45, 50, 120, 25, 24, "strategic"),
    Player("P3", "archer", 50, 35, 95, 60, 22, 20, "aggressive"),
    Player("P4", "tank", 55, 40, 60, 100, 28, 22, "strategic"),
]

# Instanciar el modelo con los datos de los jugadores
model = GameModel(players_data)

# Entrenar los modelos
model.train_classification_model()
model.train_regression_model()
model.train_clustering_model()

# Crear un nuevo jugador para realizar predicciones
new_player = Player("TestPlayer", "mage", 42, 33, 88, 45, 21, 0)

# Realizar predicciones
predicted_style = model.predict_style(new_player)
predicted_victories = model.predict_victories(new_player)
predicted_cluster = model.assign_cluster(new_player)
```

```
24.
25. # Imprimir los resultados de las predicciones
26. print(f"Estilo de juego predicho para {new_player.player_name}:
    {predicted_style}")
27. print(f"Victorias predichas para {new_player.player_name}:
    {predicted_victories:.2f}")
28. print(f"Cluster asignado a {new_player.player_name}: {predicted_cluster}")
```

Salida esperada

```
1. Estilo de juego predicho para TestPlayer: aggressive
2. Victorias predichas para TestPlayer: 17.70
3. Cluster asignado a TestPlayer: 0
```

Tarea Opcional: Mostrar jugadores por cluster

Para este ejercicio adicional, te proponemos una función que te permitirá visualizar los jugadores agrupados por clusters. Esta función será útil para explorar y entender cómo el modelo de clustering (KMeans) ha agrupado a los jugadores en función de sus características.

Descripción de la tarea:

Debes implementar una función que imprima los jugadores asignados a cada cluster después de que el modelo KMeans haya sido entrenado. Cada cluster debe ser visualizado con el nombre del jugador, el tipo de personaje y su estilo de juego.

Especificaciones:

- Utiliza el modelo `KMeans` entrenado en el ejercicio anterior.
- La función debe recorrer los diferentes clusters y mostrar los jugadores pertenecientes a cada uno, con los siguientes detalles:
  - Nombre del jugador
  - Tipo de personaje
  - Estilo de juego
- La salida debe ser algo como:

```
1. Cluster 0:
2. P1 - Mage - Aggressive
3. P3 - Archer - Aggressive
4. Cluster 1:
5. P2 - Tank - Strategic
6. P4 - Tank - Strategic
```

Consejos:

- Puedes utilizar `model.cluster_model.labels_` para obtener las asignaciones de los clusters.
- Convierte los datos de los jugadores en un `DataFrame` para facilitar la manipulación y visualización de la información.
- La función debe imprimir los jugadores por cada cluster, y para ello puedes agrupar los jugadores según el valor de `model.cluster_model.labels_`.

53. Predicción del consumo energético
□ Contexto:

La eficiencia energética es una prioridad en las ciudades modernas. Las compañías eléctricas intentan predecir cuánto se consumirá en función de las condiciones meteorológicas. En este proyecto, desarrollarás un modelo de **regresión lineal** que permita predecir el consumo de energía en función de la **temperatura ambiental**.

□ Objetivo del proyecto:

Construir un sistema que:

1. Genere datos sintéticos con `numpy` representando temperatura (°C) y consumo energético (kWh).
2. Use regresión lineal (`sklearn.linear_model.LinearRegression`) para aprender la relación entre ambas variables.
3. Permita hacer predicciones para nuevas temperaturas.
4. Visualice los datos y el modelo con `matplotlib`.

□ **Requerimientos:**

1. Crear una clase `EnergyRecord`

- Guarda los atributos: `temperature` y `consumption`.
- Añade un método `.to_vector()` que devuelva `[temperature]` como vector de entrada al modelo.

2. Generar los datos con una clase `EnergyDataGenerator`

- Crea datos sintéticos con `numpy.random.uniform(-5, 35)` para la temperatura.
- Calcula el consumo simulando que **cuando hace más frío o más calor que 20 °C**, el consumo aumenta:
    - consumo = 100 + (abs(temperatura - 20) * 3) + ruido
- Añade un poco de `ruido` con `numpy.random.normal(0, 5)`.

3. Crear la clase `EnergyRegressor`

- Usa `LinearRegression` de `sklearn` para ajustar el modelo.
- Métodos necesarios:
    - `fit()` para entrenar con una lista de `EnergyRecord`.
    - `predict(temperature)` para predecir consumo dado una temperatura.
    - `get_model()` para acceder al modelo (útil para graficar).

4. Implementar una clase `EnergyPredictionExample`

- Que cree los datos, entrene el modelo y prediga para una temperatura nueva (por ejemplo, 30 °C).
- También debe mostrar una gráfica:
    - Un `scatter plot` de los datos.
    - Una línea roja representando la recta de regresión.

5. Visualización con `matplotlib`

- Agrega títulos, etiquetas de ejes y leyenda para una mejor comprensión.
- Usa `.plot()` para la línea de predicción del modelo.

□ Ejemplo de uso

```
example = EnergyPredictionExample()
example.run()
```

Salida esperada

```
□ Temperatura: 30 °C
□ Predicción de consumo: 120.70 kWh
```