**INTRODUCCIÓN AL UNITTEST**

**¿Qué es unittest?**

unittest es un marco de pruebas inspirado en el framework JUnit para Java y se utiliza para escribir y ejecutar pruebas automáticas en Python. Permite a los desarrolladores verificar que las partes individuales de su código (unidades) funcionen según lo esperado.

**Características principales**

1. **Organización de Pruebas**: Las pruebas se agrupan en clases derivadas de unittest.TestCase.
2. **Métodos de Configuración y Desmontaje**: Proporciona métodos como setUp() y tearDown() para preparar el entorno de prueba y limpiarlo después de cada prueba.
3. **Asserts**: Incluye múltiples métodos de aserción (assertEqual, assertTrue, assertFalse, etc.) para verificar condiciones específicas en las pruebas.
4. **Descubrimiento de Pruebas**: Permite la búsqueda automática de casos de prueba en un directorio.
5. **Informes**: Genera informes detallados sobre el resultado de las pruebas, incluyendo cuáles pasaron, fallaron o tuvieron errores.

**Ejemplo Básico**

Aquí tienes un ejemplo sencillo de cómo usar unittest:

python

Copiar código

import unittest

def suma(a, b):

return a + b

class TestSuma(unittest.TestCase):

def setUp(self):

# Preparación antes de cada prueba

self.a = 10

self.b = 20

def tearDown(self):

# Limpieza después de cada prueba

pass

def test\_suma(self):

# Prueba del método suma

self.assertEqual(suma(self.a, self.b), 30)

def test\_suma\_negative(self):

self.assertEqual(suma(-1, -1), -2)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Cómo Ejecutar las Pruebas**

Para ejecutar las pruebas, puedes simplemente ejecutar el script Python que contiene las pruebas. Si estás utilizando un entorno de desarrollo integrado (IDE) como PyCharm o VSCode, estos generalmente tienen soporte integrado para ejecutar pruebas unitarias.

**Conclusión**

unittest es una herramienta poderosa y flexible para asegurarte de que tu código Python funcione correctamente mediante la automatización de pruebas. A medida que tu proyecto crece, las pruebas unitarias te ayudan a mantener la calidad y a detectar errores de forma temprana.

**UNIT TEST PARA DATA SCIENTISTS**

En el contexto del trabajo de un data scientist, la librería unittest se utiliza principalmente para asegurar que el código que manipula datos, entrena modelos y realiza análisis funcione correctamente y produzca resultados consistentes. Aquí hay algunas formas específicas en que un data scientist puede utilizar unittest:

**1. Validación de Preprocesamiento de Datos**

El preprocesamiento de datos es una etapa crucial en cualquier proyecto de machine learning. Con unittest, un data scientist puede escribir pruebas para asegurar que las funciones de limpieza y transformación de datos funcionen como se espera.

python

Copiar código

import unittest

import pandas as pd

def eliminar\_nulos(df):

return df.dropna()

class TestPreprocesamiento(unittest.TestCase):

def test\_eliminar\_nulos(self):

data = {'col1': [1, 2, None], 'col2': [4, None, 6]}

df = pd.DataFrame(data)

result = eliminar\_nulos(df)

self.assertEqual(result.shape[0], 1)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**2. Verificación de Funciones de Ingeniería de Características**

Las funciones de ingeniería de características suelen ser complejas y es importante asegurarse de que produzcan las características correctas.

python

Copiar código

def agregar\_caracteristica(df):

df['nueva\_col'] = df['col1'] \* df['col2']

return df

class TestIngenieriaCaracteristicas(unittest.TestCase):

def test\_agregar\_caracteristica(self):

data = {'col1': [1, 2], 'col2': [3, 4]}

df = pd.DataFrame(data)

result = agregar\_caracteristica(df)

self.assertTrue('nueva\_col' in result.columns)

self.assertEqual(result['nueva\_col'].iloc[0], 3)

self.assertEqual(result['nueva\_col'].iloc[1], 8)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**3. Evaluación de Modelos de Machine Learning**

Se pueden escribir pruebas para asegurarse de que los modelos se entrenen correctamente y produzcan resultados dentro de un rango esperado.

python

Copiar código

from sklearn.linear\_model import LinearRegression

from sklearn.datasets import make\_regression

def entrenar\_modelo(X, y):

modelo = LinearRegression()

modelo.fit(X, y)

return modelo

class TestModelos(unittest.TestCase):

def test\_entrenar\_modelo(self):

X, y = make\_regression(n\_samples=100, n\_features=1, noise=0.1)

modelo = entrenar\_modelo(X, y)

self.assertIsNotNone(modelo)

self.assertGreaterEqual(modelo.score(X, y), 0.9)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**4. Verificación de Resultados de Análisis**

Los análisis y visualizaciones de datos también pueden ser probados para asegurar que producen resultados consistentes.

python

Copiar código

import numpy as np

def calcular\_media(arr):

return np.mean(arr)

class TestAnalisisDatos(unittest.TestCase):

def test\_calcular\_media(self):

arr = np.array([1, 2, 3, 4, 5])

self.assertEqual(calcular\_media(arr), 3)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Beneficios del Uso de unittest para un Data Scientist**

1. **Consistencia**: Ayuda a asegurar que el código funcione de manera consistente, lo cual es crucial cuando se manejan grandes cantidades de datos.
2. **Mantenimiento**: Facilita el mantenimiento del código y la detección de errores al realizar cambios en el código base.
3. **Confianza**: Incrementa la confianza en el código, especialmente en proyectos colaborativos, donde múltiples personas trabajan en el mismo código.
4. **Documentación**: Las pruebas pueden servir como una forma de documentación para que otros desarrolladores entiendan cómo se espera que funcione el código.

En resumen, unittest es una herramienta valiosa para los data scientists para garantizar la fiabilidad y estabilidad de sus scripts y pipelines de datos.

4o

Aquí tienes un ejemplo de cómo puedes usar unittest para probar el entrenamiento de un modelo de machine learning utilizando la biblioteca scikit-learn. En este ejemplo, vamos a entrenar un modelo de regresión lineal y verificar que el modelo se entrene correctamente y que su rendimiento sea aceptable.

**Ejemplo Completo**

**1. Código del Modelo y Preprocesamiento**

Primero, definimos las funciones para generar datos, preprocesarlos y entrenar el modelo.

python

Copiar código

from sklearn.linear\_model import LinearRegression

from sklearn.datasets import make\_regression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

import numpy as np

def generar\_datos(n\_samples=100, n\_features=1, noise=0.1):

X, y = make\_regression(n\_samples=n\_samples, n\_features=n\_features, noise=noise)

return X, y

def dividir\_datos(X, y, test\_size=0.2, random\_state=42):

return train\_test\_split(X, y, test\_size=test\_size, random\_state=random\_state)

def entrenar\_modelo(X\_train, y\_train):

modelo = LinearRegression()

modelo.fit(X\_train, y\_train)

return modelo

def evaluar\_modelo(modelo, X\_test, y\_test):

y\_pred = modelo.predict(X\_test)

mse = mean\_squared\_error(y\_test, y\_pred)

return mse

**2. Código de Pruebas con unittest**

A continuación, escribimos las pruebas unitarias para verificar el correcto funcionamiento de las funciones anteriores.

python

Copiar código

import unittest

class TestModeloMachineLearning(unittest.TestCase):

def setUp(self):

# Generamos los datos y los dividimos en conjuntos de entrenamiento y prueba

X, y = generar\_datos(n\_samples=100, n\_features=1, noise=0.1)

self.X\_train, self.X\_test, self.y\_train, self.y\_test = dividir\_datos(X, y)

def test\_generar\_datos(self):

# Verificamos que la función generar\_datos crea las dimensiones correctas

X, y = generar\_datos(n\_samples=100, n\_features=1, noise=0.1)

self.assertEqual(X.shape[0], 100)

self.assertEqual(X.shape[1], 1)

self.assertEqual(y.shape[0], 100)

def test\_dividir\_datos(self):

# Verificamos que los datos se dividen correctamente

X\_train, X\_test, y\_train, y\_test = dividir\_datos(self.X\_train, self.y\_train)

self.assertEqual(len(X\_train), 80)

self.assertEqual(len(X\_test), 20)

self.assertEqual(len(y\_train), 80)

self.assertEqual(len(y\_test), 20)

def test\_entrenar\_modelo(self):

# Verificamos que el modelo se entrena sin errores y es un objeto de tipo LinearRegression

modelo = entrenar\_modelo(self.X\_train, self.y\_train)

self.assertIsInstance(modelo, LinearRegression)

def test\_evaluar\_modelo(self):

# Entrenamos el modelo y verificamos que el MSE es razonable

modelo = entrenar\_modelo(self.X\_train, self.y\_train)

mse = evaluar\_modelo(modelo, self.X\_test, self.y\_test)

self.assertGreaterEqual(mse, 0)

self.assertLessEqual(mse, 10) # Asumiendo que el error debe ser menor que 10

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Explicación del Código**

1. **Funciones del Modelo**:
   * generar\_datos(): Genera datos de regresión sintéticos.
   * dividir\_datos(): Divide los datos en conjuntos de entrenamiento y prueba.
   * entrenar\_modelo(): Entrena un modelo de regresión lineal.
   * evaluar\_modelo(): Evalúa el modelo usando el error cuadrático medio (MSE).
2. **Clases de Pruebas con unittest**:
   * **setUp()**: Configura los datos necesarios antes de cada prueba.
   * **test\_generar\_datos()**: Verifica que la generación de datos funcione correctamente.
   * **test\_dividir\_datos()**: Verifica que los datos se dividan correctamente.
   * **test\_entrenar\_modelo()**: Verifica que el modelo se entrene correctamente.
   * **test\_evaluar\_modelo()**: Verifica que la evaluación del modelo produzca un MSE razonable.

**Ejecución de las Pruebas**

Para ejecutar las pruebas, simplemente ejecuta el script en el que has definido las pruebas. El marco unittest se encargará de descubrir y ejecutar todas las pruebas definidas.

Este enfoque garantiza que cada componente de tu pipeline de machine learning funcione correctamente y te permite detectar problemas rápidamente, mejorando así la confiabilidad de tu código.

El error que estás viendo indica que la prueba test\_crear\_nueva\_caracteristica falló porque el valor esperado no coincide con el valor obtenido para feature\_sum. Esto sugiere que el cálculo del valor esperado no está reflejando correctamente la lógica de tu aplicación o la manipulación de datos.

Vamos a revisar el código para asegurarnos de que el cálculo de expected\_sum esté correcto y refleje lo que realmente se espera después de crear la nueva característica feature\_sum.

**Código Ajustado**

Asegúrate de que el cálculo de expected\_sum esté alineado con la lógica de tu función crear\_nueva\_caracteristica. Aquí te doy un ejemplo ajustado donde suponemos que crear\_nueva\_caracteristica suma las columnas feature1 y feature2 para generar feature\_sum.

python

Copiar código

import unittest

import pandas as pd

import numpy as np

# Función para crear una nueva característica

def crear\_nueva\_caracteristica(df):

df['feature\_sum'] = df['feature1'] + df['feature2']

return df

# Clase de pruebas unitarias

class TestPipelineMachineLearning(unittest.TestCase):

def setUp(self):

# Crear un DataFrame de ejemplo

data = {

'feature1': [1, 2, 3, 4, 5, np.nan],

'feature2': [2, 3, 4, 5, 6, 7],

'target': [1.2, 2.4, 3.5, 4.8, 5.0, 6.1]

}

self.df = pd.DataFrame(data)

self.target\_column = 'target'

# Imputar nulos para las pruebas

self.df\_imputed = imputar\_nulos(self.df)

def test\_crear\_nueva\_caracteristica(self):

df\_new = crear\_nueva\_caracteristica(self.df\_imputed)

self.assertTrue('feature\_sum' in df\_new.columns)

# Calcular el valor esperado basado en los datos imputados

expected\_sum = self.df\_imputed['feature1'].iloc[0] + self.df\_imputed['feature2'].iloc[0]

self.assertEqual(df\_new['feature\_sum'].iloc[0], expected\_sum)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

**Explicación del Cambio**

1. **Función crear\_nueva\_caracteristica**: Se asume que esta función suma las columnas feature1 y feature2 para crear feature\_sum.
2. **test\_crear\_nueva\_caracteristica**:
   * Se calcula expected\_sum como la suma de feature1 y feature2 del DataFrame original (self.df\_imputed).
   * Se utiliza self.assertEqual para comparar el valor calculado de feature\_sum con expected\_sum.

Este enfoque asegura que la prueba test\_crear\_nueva\_caracteristica verifique correctamente que la función crear\_nueva\_caracteristica esté produciendo el resultado esperado.

Si la lógica de crear\_nueva\_caracteristica es diferente (por ejemplo, si se requiere una suma ponderada u otra operación), ajusta el cálculo de expected\_sum en consecuencia para reflejar esa lógica específica.