Programación orientada a objetos

Introducción


REDES NEURONALES
ESTUDIANTES DE LA DIPLO
REDES NEURONALES
PROGRAMACIÓN ORIENTADA A OBJETOS
ESTUDIANTES DE LA DIPLO
imgflip.com

Programación orientada a objetos

La programación orientada a objetos es el **paradigma de programación** más popular utilizado para el desarrollo de software basado en la creación de **"objetos" reutilizables** que tienen sus propias propiedades y comportamiento sobre los que se puede *actuar, manipular y agrupar*.

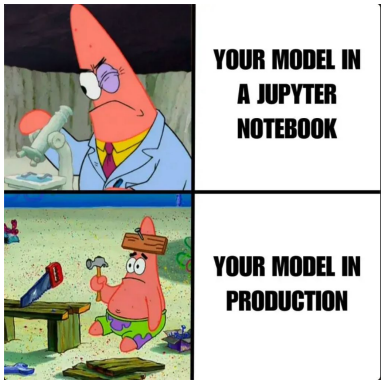
Se utiliza para estructurar un programa de software en piezas simples y reutilizables de planos de código (generalmente llamados **clases**), que a su vez se utilizan para crear instancias individuales de **objetos**.

Algunas ventajas de la POO

- Permite modelar el código de una forma más cercana a como vemos los objetos y sus relaciones en la realidad.
- Permite reusar el código porque una clase se puede definir una vez y reutilizarse muchas veces.
- Por estas razones, el código sea más reutilizable y mantenible.

Algunas ventajas de la POO

- Muy importante para el escalado de notebooks a sistemas productivos.



Conceptos importantes

CLASES

Una **clase** es un **modelo abstracto** que crea objetos, instancias concretas de la misma.

Las clases suelen representar categorías con **atributos** comunes.

También definen funciones (**métodos**) disponibles para tales objetos.

```
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print("Tipo de df:", type(df))
print("Contenido de df:\n", df)
```

```
Tipo de df: <class 'pandas.core.frame.DataFrame'>
Contenido de df:
    A  B
0  1  4
1  2  5
2  3  6
```

```
[16] df.shape
```

```
(3, 2)
```

```
[17] df.columns
```

```
Index(['A', 'B'], dtype='object')
```

Conceptos importantes

OBJETOS

Un **objeto** **es una instancia de una clase**,

- Toma valores concretos para sus **atributos**.
- Usa **métodos** comunes para definir comportamiento particular.

```
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
print("Tipo de df:", type(df))
print("Contenido de df:\n", df)
```

```
Tipo de df: <class 'pandas.core.frame.DataFrame'>
Contenido de df:
    A  B
0  1  4
1  2  5
2  3  6
```

```
df2 = pd.DataFrame({'Col1': [4, 5, 6, 'a']})
```

Conceptos importantes

MÉTODOS

- Funciones similares a las clásicas *def*...
- Solo que se aplican normalmente[1] a un **objeto**, y define su **comportamiento**.
- Por ejemplo, un dataframe vacío se comportará distinto que uno con datos.

```
[27] df.max(axis=0)  # método 1
```

	0
A	3
B	6

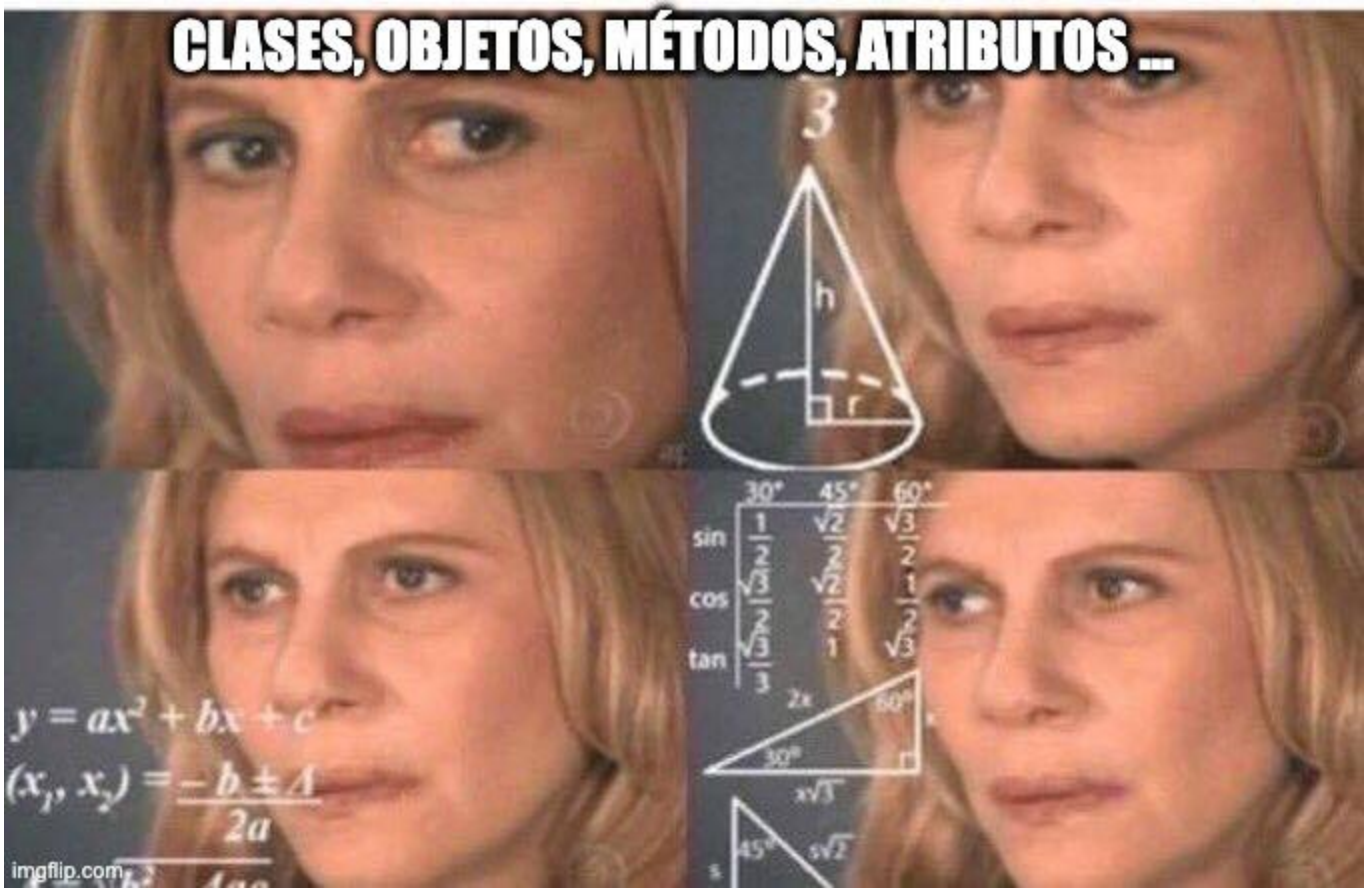
dtype: int64

```
[30] df.sort_values(by='A')  # método 2
```

	A	B
0	1	4
1	2	5
2	3	6

```
sum([1,2,3])  # función "sin objeto" (no es método)
```

```
6
```


CLASES, OBJETOS, MÉTODOS, ATRIBUTOS ...
3
h
r
30° 45° 60°
sin 1/2 √2/2 √3/2
cos √3/2 √2/2 1/2
tan √3/3 1 √3
2x
60°
30°
x√3
45°
x√2
$y = ax^2 + bx + c$
$(x_1, x_2) = \frac{-b \pm \Delta}{2a}$
imgflip.com

Ejemplo de modelo con POO

El método de inicialización

Método especial que inicia la vida de un objeto, creando una instancia

En python, tiene un nombre predefinido, `__init__`, y no tiene un valor de retorno

```
class EsqueletoKNN:
    def __init__(self, k=3):
        self.k = k
```

El método de inicialización

Un método de inicialización debe aceptar el parámetro especial `self`.

Los **atributos** del objeto toman valor dentro del método.

El parámetro `self` permite que el método de inicialización seleccione la instancia del objeto.

```
class EsqueletoKNN:
    def __init__(self, k=3):
        self.k = k
```

Ejemplo de modelo

Consideremos una implementación de un KNN con POO

```
class ModeloKNN:
    def __init__(self, k=3):
      self.k = k

    def fit(self, X, y): # fit es un método
      self._X_train = X  # "entrenamiento"
      self._y_train = y
      return self

    def predict_knn(self, X):
      # WIP
      pass
```

```
knn1 = ModeloKNN()
knn1.fit([[1, 2], [1.5, 1.8]], [0, 0])
knn1.k
```

```
3
```

```
knn2 = ModeloKNN(k=4)
knn2.fit([[2, 2.5], [2.5, 3]], [0, 0])
knn2.k
```

```
4
```

Sin POO podría ser algo así...

```
knn1_k = 3
knn2_k = 4

# Definiríamos los valores de "entrenamiento"
# para cada modelo
X_knn_1 = np.array([[1, 2], [1.5, 1.8]])
y_knn_1 = np.array([0, 0])

X_knn_2 = np.array([[2, 2.5], [2.5, 3]])
y_knn_2 = np.array([0, 0])

def predict_knn(k, X_knn, y_knn):
  # WIP
  pass
```

Ejemplo de modelo

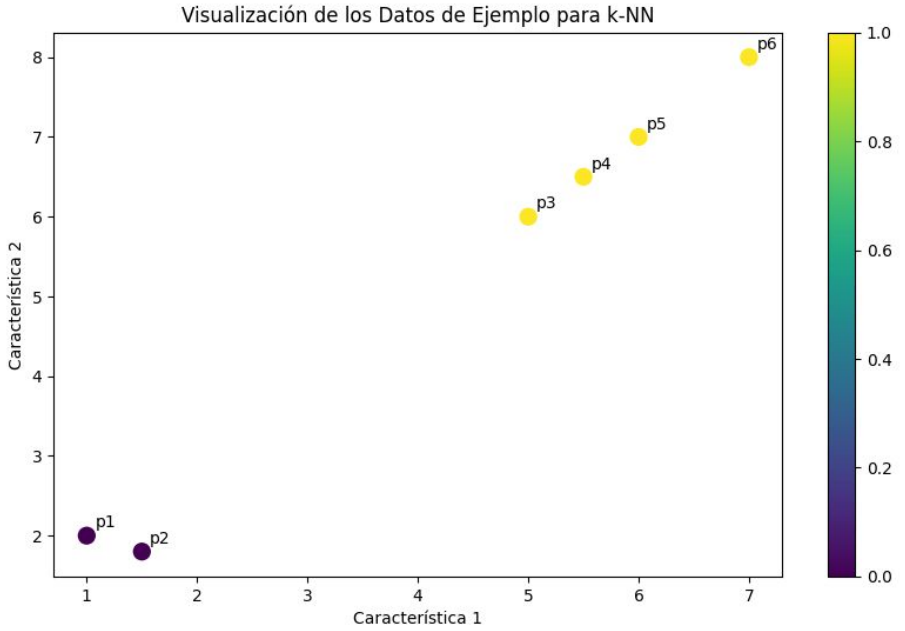
Consideremos una implementación de un KNN con POO

```
def predict(self, X):
    predicciones = []
    for x in X:
        distancias = [np.sqrt(np.sum((x - x_train)**2)) for x_train in self.X_train]  # distancia euclidiana
        k_indices = np.argsort(distancias)[:self.k]  # argsort devuelve los índices ordenados
        k_vecinos = [self.y_train[i] for i in k_indices]  # usamos los índices para obtener los valores de y_train
        prediccion = max(set(k_vecinos), key=k_vecinos.count)  # obtenemos la moda
        predicciones.append(prediccion)  # agregamos la predicción a la lista
    return np.array(predicciones)
```

Ejemplo de modelo

Consideremos una implementación de un KNN con POO

```
# Crear datos de ejemplo
X = np.array([
    [1, 2],
    [1.5, 1.8],
    [2, 2.5],
    [2.5, 3],
    [5, 6],
    [5.5, 6.5],
    [6, 7],
    [7, 8]
])
y = np.array([0, 0, 0, 0, 1, 1, 1, 1])
```



Ejemplo de modelo

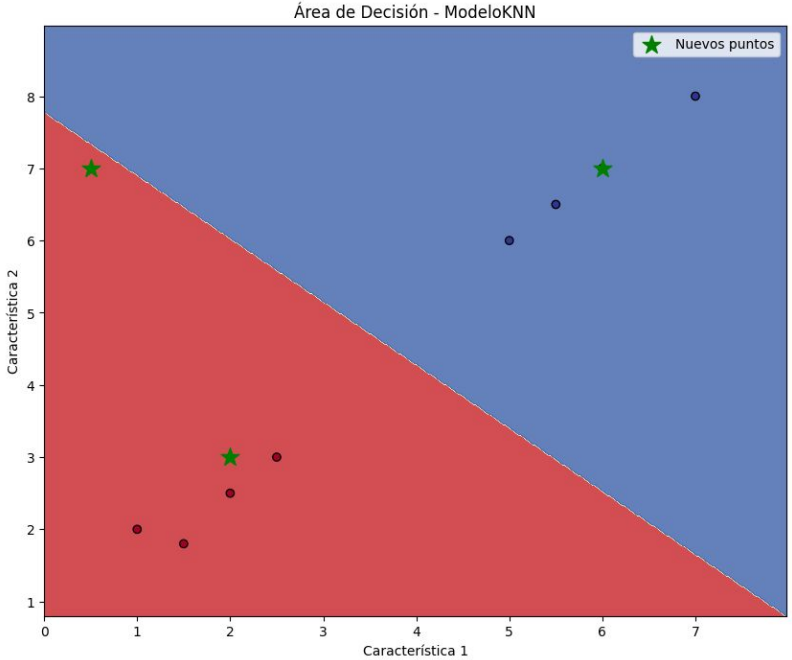
Consideremos una implementación de un KNN con POO

```
# Se instancia y entrena el modelo
modelo_knn = ModeloKNN(k=3)
modelo_knn.fit(X, y)
```

```
<__main__.ClasificadorSimple at 0x7c660fac76d0>
```

Ejemplo de modelo

Consideremos una implementación de un KNN con POO



Agreguemos herencia!

La herencia permite que una nueva clase asuma las propiedades y comportamientos de otra clase.

La clase de la que se hereda se denomina ***clase padre.*** Cualquier clase que hereda de una clase principal se denomina ***clase hija***.

Por ejemplo, podríamos crear un modelo *Minkowski* (clase hija) a partir de *ModeloKNN* (clase padre).

Las clases secundarias o hijas no solo heredan todas las propiedades y métodos, sino que también pueden expandirlos o sobreescribirlos.

```
class ModeloKNNMinkowski(ModeloKNN):
    def __init__(self, k=3, p=2):  # se agrega p, parámetro de la distancia de Minkowski
        super().__init__(k)  # Llamada al constructor de la clase padre
        self._p = p  # Parámetro para la distancia de Minkowski
```

Ejemplo de modelo con POO

Implementamos ModeloKNNMinkowski, que hereda de ModeloKNN

```
class ModeloKNNMinkowski(ModeloKNN):
    def __init__(self, k=3, p=2):  # se agrega p, parámetro de la distancia de Minkowski
        super().__init__(k)  # Llamada al constructor de la clase padre
        self.p = p  # Parámetro para la distancia de Minkowski

    def predict(self, X):  # Sobreescritura del método predict padre
        predicciones = []
        for x in X:
            # Cálculo de la distancia de Minkowski
            distancias = [np.sum(np.abs(x - x_train)**self.p)**(1/self.p) for x_train in self.X_train]
            k_indices = np.argsort(distancias)[:self.k]
            k_vecinos = [self.y_train[i] for i in k_indices]
            prediccion = max(set(k_vecinos), key=k_vecinos.count)
            predicciones.append(prediccion)
        return np.array(predicciones)
```

Ejemplo de modelo con POO

Implementamos ModeloKNNMinkowski, que hereda de ModeloKNN

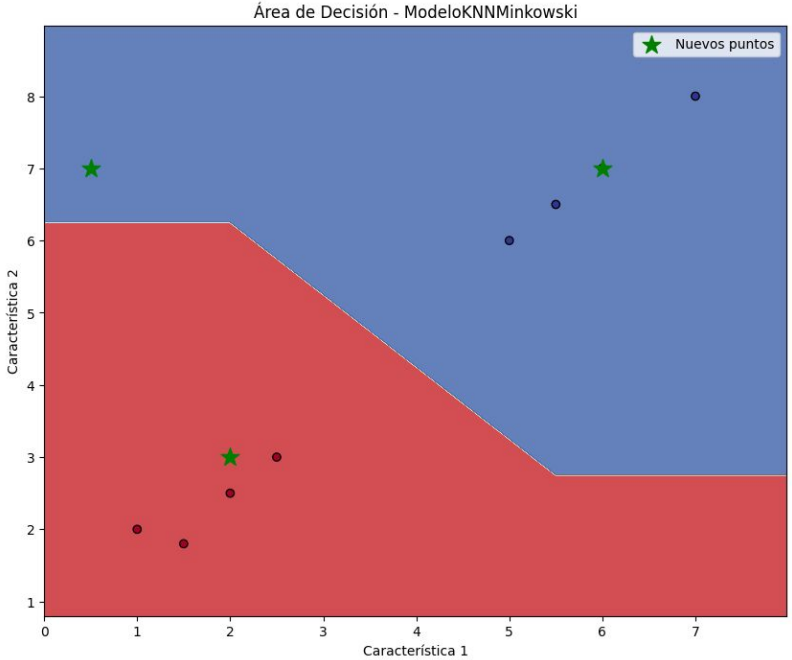
```
# Usar los mismos datos de ejemplo
modelo_knn_minkowski = ModeloKNNMinkowski(k=3, p=1)
modelo_knn_minkowski.fit(X, y)  # se usa el fit definido en ModeloKNN

predicciones_minkowski = modelo_knn_minkowski.predict(X_nuevo)
predicciones_minkowski
```

```
array([0, 1, 1])
```

Ejemplo de modelo con POO

Implementamos ModeloKNNMinkowski, que hereda de ModeloKNN



Otras propiedades fundamentales de POO

Otras propiedades...

Abstracción

Nos permite usar modelo KNN sin conocer sus detalles internos

```
# imaginemos por ejemplo que lo importamos desde otra librería...
# import ModeloKNN

modelo = ModeloKNN(k=3)
modelo.fit(X, y)
prediccion = modelo.predict([[3, 4]])
prediccion
```

```
array([0])
```

Otras propiedades...

Encapsulamiento

Los detalles internos de la clase se "protegen" de ser accedidos. Ej.: no accedemos a (X, y) sino con un método

Esta protección es por diseño.

Esto es, invocar métodos protegidos es posible[1], pero puede traer consecuencias inesperadas[2]

```
# el _ de _X_train marca que es de uso dentro
# de la clase. El atributo (o método)
# no fue pensado para accederse así
modelo._X_train
```

```
array([[1. , 2. ],
       [1.5, 1.8],
       [2. , 2.5],
       [2.5, 3. ],
       [5. , 6. ],
       [5.5, 6.5],
       [6. , 7. ],
       [7. , 8. ]])
```

Otras propiedades...

Polimorfismo

Permite que objetos de diferentes clases son tratados como objetos de una clase común, por ejemplo mediante herencia

Esto hace que podamos usar objetos de tales clases de manera uniforme, sin tener que conocer de qué clase provienen

```
for i, modelo in enumerate(modelos, 1):
    modelo.fit(X, y)
    prediccion = modelo.predict([[0.3, 7.1]])
    print(f"Predicción del Modelo {i}\
            para [0.3, 7.1]: {prediccion}")
```

```
Predicción del Modelo 1            para [0.3, 7.1]: [0]
Predicción del Modelo 2            para [0.3, 7.1]: [1]
```

LET’S CODE

3. Mi primera red neuronal: Creando y entrenando modelos con pytorch

LET'S CODE - Anexo

Introducción a Programación Orientada a Objetos