UNIDAD 1

Tema 1:

Desarrollo de Pensamiento Algorítmico mediante Actividades Prácticas

Introducción

El pensamiento algorítmico es fundamental en la ciencia de la computación y la programación. Consiste en la habilidad para formular soluciones paso a paso a problemas complejos, utilizando un conjunto de instrucciones claras y lógicas. En esta clase, exploraremos diferentes técnicas y ejercicios prácticos diseñados para desarrollar y mejorar esta habilidad.

Importancia del Pensamiento Algorítmico

El pensamiento algorítmico es esencial por varias razones:

- Permite abordar problemas complejos dividiéndolos en pasos más pequeños y manejables.
- Ayuda a mejorar la eficiencia y la precisión en la resolución de problemas.
- Es fundamental para la programación y el desarrollo de software, donde cada función y proceso se basa en algoritmos bien definidos.

Objetivos de la Clase

1. Familiarizarse con los conceptos básicos del pensamiento algorítmico.
2. Practicar la creación y análisis de algoritmos simples.
3. Aplicar técnicas de resolución de problemas a través de ejercicios prácticos.

Actividades Prácticas

Durante esta clase, trabajarás en una serie de ejercicios diseñados para desarrollar y fortalecer tu capacidad para pensar de manera algorítmica. Estos ejercicios cubrirán desde problemas simples hasta desafíos más complejos, permitiéndote aplicar los conceptos aprendidos y mejorar tu habilidad para diseñar soluciones eficientes.

Recomendaciones

- Asegúrate de entender completamente cada ejercicio antes de comenzar a escribir el código.
- Utiliza pseudocódigo o diagramas de flujo para planificar tus algoritmos antes de implementarlos en un lenguaje de programación.
- Practica la depuración y la optimización de tus algoritmos para mejorar su eficiencia y funcionalidad.

Tema 2: Debugging

Debugging

Paso 1: Instalación de extensiones
Abre Visual Studio Code. Ve a la pestaña de extensiones en la barra lateral izquierda (el icono de cuadrados apilados) o presiona Ctrl+Shift+X. Busca "Python" en el mercado de extensiones. Instala la extensión oficial de Python proporcionada por Microsoft.

Paso 2: Colocar puntos de interrupción
Abre el archivo de Python en el que deseas depurar. Haz clic en la columna a la izquierda del número de línea donde deseas establecer un punto de interrupción. Debería aparecer un punto rojo, indicando que se ha establecido un punto de interrupción.

Paso 3: Iniciar el depurador
Ve al menú de Visual Studio Code y selecciona Run > Start Debugging, o simplemente presiona F5. Esto iniciará el depurador y ejecutará tu código hasta que alcance el primer punto de interrupción. La ejecución se detendrá en el punto de interrupción y podrás inspeccionar el estado de tu programa.

Paso 4: Navegación y control del depurador
Una vez que el depurador se ha detenido en un punto de interrupción, puedes hacer varias cosas:

- **Continuar la ejecución:** Puedes continuar ejecutando el código hasta el próximo punto de interrupción o hasta que termine, utilizando el botón "Continuar" en la barra de herramientas de depuración o presionando F5.
- **Paso a paso:** Puedes ejecutar el código línea por línea, paso a paso, utilizando los botones "Paso a paso" en la barra de herramientas de depuración o las teclas de acceso directo asociadas (F10 para paso a paso, F11 para paso a paso en la función actual).
- **Inspeccionar variables:** Mientras estás en el modo de depuración, puedes ver los valores de las variables locales y globales en la ventana "Variables" en la barra lateral izquierda.
- **Modificar variables:** En algunos casos, puedes modificar el valor de las variables durante la depuración para probar diferentes escenarios.

Paso 5: Finalizar la depuración
Una vez que hayas terminado de depurar, puedes salir del modo de depuración haciendo clic en el botón "Detener" en la barra de herramientas de depuración o seleccionando Run > Stop Debugging en el menú.

Tema 3: Instalación de Python

¿Qué es Python?

Python es un lenguaje de programación potente y fácil de aprender. Tiene estructuras de datos de alto nivel eficientes y un simple pero efectivo sistema de programación orientado a objetos. La elegante sintaxis de Python y su tipado dinámico, junto a su naturaleza interpretada, lo convierten en un lenguaje ideal para scripting y desarrollo rápido de aplicaciones en muchas áreas, para la mayoría de plataformas.

En esta unidad se desarrollarán los conceptos básicos y las funcionalidades del lenguaje de programación Python y su sistema. Ayuda tener un intérprete de Python accesible para una experiencia práctica; todos los ejemplos son auto-contenidos, permitiendo utilizar el tutorial sin conexión.

Para poder empezar con Python, es necesario instalarlo. Puede descargarlo desde la página oficial de Python: https://www.python.org/downloads.

Para la instalación, siga los siguientes pasos:

Instalación de Python

- Acceder a la página oficial de Python: https://www.python.org/.
- Hacer clic en "Downloads" y en el menú hacer clic en el botón donde dice la última versión de “Python 3.12.2”.

Una vez descargado, se deberá hacer doble clic en el ejecutable.
a. Seleccionar las dos opciones que aparecen debajo:
css
Copiar código

```
  I. La primera es para dar privilegios al ejecutable.
  II. La segunda es para añadir a las variables de entorno la ruta en donde se instalará Python, lo cual nos permitirá poder ejecutarlo desde cualquier parte del sistema operativo.
```

- b. A continuación, hacer clic en la opción de “Install Now”. Aparecerá una ventana que nos preguntará si queremos ejecutarlo y le damos que sí.
- Si todo se instaló correctamente aparecerá la siguiente ventana:
  a. Hacer clic en la opción “Disable path length limit”. Esto nos permitirá trabajar con rutas de archivos largas en Windows, ya que hay una limitación en la longitud de la ruta de archivo que puede causar problemas al trabajar con rutas muy largas. A continuación, aparecerá una ventana de confirmación y le damos que sí.
- Al finalizar, nos debería quedar una ventana indicando el éxito de la instalación.

- Para comprobar que Python se instaló correctamente, debemos abrir una consola de comandos y ejecutar lo siguiente: “python --version”. Esto nos debería mostrar la versión de Python que instalamos.

Tema 4: Módulos y Librerías

Módulos

Un módulo o module en Python es un fichero `.py` que alberga un conjunto de funciones, variables o clases y que puede ser usado por otros módulos. Nos permiten reutilizar código y organizarlo mejor en namespaces. Por ejemplo, podemos definir un módulo `mimodulo.py` con dos funciones `suma()` y `resta()`.

```
# mimodulo.py
def suma(a, b):
    return a + b
def resta(a, b):
    return a - b
```

Una vez definido, dicho módulo puede ser usado o importado en otro fichero usando import para importar todo el contenido.

```
# otromodulo.py
import mimodulo
print(mimodulo.suma(4, 3)) # 7
print(mimodulo.resta(10, 9)) # 1
```

También podemos importar únicamente los componentes que nos interesen:

```
from mimodulo import suma, resta
print(suma(4, 3)) # 7
print(resta(10, 9)) # 1
```

Por último, podemos importar todo el módulo con `*`, sin necesidad de usar `mimodulo.`

```
from mimodulo import *
print(suma(4, 3)) # 7
print(resta(10, 9)) # 1
```

Cambiando los Nombres con `as`

Es posible cambiar el nombre del módulo usando `as`.

```
import moduloconnombrelargo as m

print(m.hola)
```

Librerías

Las bibliotecas (o librerías) de Python son colecciones de código predefinido que proporcionan funcionalidades específicas para realizar diversas tareas.

- **Amplia gama de funcionalidades:** Python cuenta con una amplia gama de bibliotecas.
- **Comunidad activa:** La comunidad de Python es muy activa.
- **Fácil instalación y uso:** La mayoría de las bibliotecas se pueden instalar con `pip`.

Gestor de paquetes PIP

PIP (Python Package Index) permite instalar, actualizar y eliminar paquetes de Python de forma sencilla.

```
pip --version

pip3 --version
```

Para instalar un paquete:

```
pip install <module_name>
```

Para actualizar un paquete:

```
pip install <module_name> --upgrade
```

Para eliminar un paquete:

```
pip uninstall <module_name>
```

Listar paquetes instalados:

```
pip list
```

Mostrar información sobre un paquete:

```
pip show <module_name>
```

Buscar paquetes:

```
pip search término-de-búsqueda
```

Otros gestores usados en Python

Conda

Conda es un gestor de paquetes y un sistema de gestión de entornos para Python, R y otros lenguajes.

Pipenv

Pipenv es una herramienta de gestión de dependencias y entornos para Python.

Poetry

Poetry es otra herramienta de gestión de dependencias y paquetes para Python.

Anaconda

Anaconda es una distribución de Python que incluye una amplia gama de paquetes y herramientas utilizadas en el ámbito científico.

Entornos virtuales

Los entornos virtuales son directorios de instalación aislados. Este aislamiento permite gestionar dependencias sin conflictos.

Virtualenv

virtualenv es una herramienta que se utiliza para crear entornos Python aislados. Para instalarlo usamos **pip**

```
pip install virtualenv
```

Verifica la instalación:

```
virtualenv --version
```

Para crear un entorno virtual:

```
virtualenv --no-site-packages my-env
```

Esto crea una carpeta en el directorio actual con el nombre del entorno (`my-env/`). También puedes especificar la versión de Python:

```
virtualenv --python=/usr/bin/python3.10 my-env
```

Listar entornos disponibles:

```
lsvirtualenv
```

Activar un entorno en Unix:

```
source my-env/bin/activate
```

En Windows:

```
.\my-env\Scripts\activate
```

Para instalar paquetes:

```
pip install numpy
```

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

Para crear un archivo `requirements.txt`:

```
pip freeze > requirements.txt
```

Desactivar un entorno:

```
deactivate
```

Eliminar un entorno: simplemente elimina la carpeta.

Alternativa “venv”

Venv es un módulo integrado en Python que permite crear entornos virtuales de manera nativa.

Crear un nuevo entorno virtual:

```
python -m venv my-env
```

Activar el entorno virtual en Windows:

```
.\my-env\Scripts\activate
```

Desactivar el entorno virtual:

```
deactivate
```

Tema 5: NumPy

NumPy: estructuras matriciales

NumPy es una librería de Python que posibilita el trabajar con vectores, matrices y arreglos n-dimensionales de manera eficiente. Numpy tiene una amplia gama de opciones para trabajar, entre ellas están el dominio del álgebra lineal, la transformación de Fourier y las matrices.

Los objetos de arrays en Numpy se denominan ndarray y se encuentran en el núcleo del paquete de la librería. Los ndarrays capturan arrays con n_dimensiones y diferentes tipos de datos.
Una de las grandes ventajas de los arrays de Numpy es que son bastante ordenados, ya que, a diferencia de las listas en Python (que crecen de manera dinámica), este tipo de matrices ya tienen un tamaño fijo en su creación. De modo que cambiar el tamaño de un ndarray creará una nueva matriz y eliminará la original.

Es necesario que todos los elementos de un array tengan el mismo tipo de datos, porque así tendrán el mismo tamaño en memoria. La única excepción es que se pueden tener matrices de objetos, lo cual permite las matrices de elementos de diferentes tamaños.

Las matrices de Numpy son mucho mejores que las listas en muchas ocasiones, ya que se almacenan en un lugar continuo en la memoria, a diferencia de las listas. Por tanto, durante las transacciones es mucho más rápido acceder a ellas y manipularlas. Esto es lo que se conoce como cercanía de referencias o locality of reference.

Una de las grandes ventajas de los arrays de Numpy es que permiten crear operaciones matemáticas de alta complejidad. Al mismo tiempo, se puede trabajar con ecuaciones lineales de álgebra lineal o estructuras de datos, entre otras operaciones.

¿Por qué usar Numpy?

Numpy no está escrito en lenguaje Python o, al menos, no por completo. Está escrito en Python solo parcialmente, ya que la mayoría de las partes que requieren un cálculo rápido están escritas en C o C++. Esta es una de las razones por las que Numpy es tan rápido, porque está escrito en C y este, al trabajar a bajo nivel y compilar a código máquina directamente, ofrece un gran poder.

¿Cuál es la diferencia entre arrays y matrices?

Las matrices, en un sentido estricto, son un tipo de array de dos dimensiones, ya que poseen un largo y un ancho preciso.
Los arrays, por otro lado, no tienen un número definido de dimensiones; pueden ser de n_dimensiones e ir cambiando. Esto quiere decir que pueden ser cubos, hipercubos o cualquier otro tipo de forma tridimensional, contrario a las matrices, que poseen una forma rectangular con sus filas y columnas bien definidas.

Por lo cual:

- Los arrays de una dimensión representan vectores.
- Los arrays de dos dimensiones representan matrices.
- Los arrays más grandes representan tensores.

¿Qué son los Arreglos de NumPy?

Los arreglos NumPy son la forma principal de almacenar datos utilizando la biblioteca NumPy. Son similares a las listas normales en Python, pero tienen la ventaja de ser más rápidas y tener más métodos integrados.

Los arreglos de NumPy son creados llamando al método array() de la librería de NumPy. Dentro del método, deberías pasar una lista.

Ejemplo básico:

```
import numpy as np
sample_list = [1, 2, 3]
np.array(sample_list) # array([1,2,3])
```

El contenedor array() indica que esta ya no es una lista normal de Python. En cambio, es un arreglo de NumPy.
Hay dos tipos diferentes de arreglos de NumPy: vectores y matrices.

Los vectores son arreglos de NumPy uni-dimensionales y se ve así:

```
import numpy as np
my_vector = np.array(['este', 'es', 'un', 'vector'])
```

Las matrices son arreglos bi-dimensionales y son creadas pasando una lista de lista dentro del método np.array(). Un ejemplo es el siguiente.

```
import numpy as np
my_matrix = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
np.array(my_matrix)
```

Arreglos unidimensionales (Vectores)

Para definir un arreglo unidimensional (vector), debemos pasar una lista de valores (usualmente numéricos), por ejemplo:

```
import numpy as np
x = np.array([1,2,3,5,6,7,8,9,10])
type(x) # numpy.ndarray
```

Otras funciones útiles que permiten generar arrays son:

- np.empty(dimensiones): Crea y devuelve una referencia a un array vacío con las dimensiones especificadas en la tupla dimensiones.
- np.zeros(dimensiones): Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos ceros.
- np.ones(dimensiones): Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos unos.
- np.full(dimensiones, valor): Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son todos valor.
- np.identity(n): Crea y devuelve una referencia a la matriz identidad de dimensión n.
- np.arange(inicio, fin, salto): Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia desde inicio hasta fin tomando valores cada salto.
- np.linspace(inicio, fin, n): Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia de n valores equidistantes desde inicio hasta fin.
- np.random.random(dimensiones): Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla dimensiones cuyos elementos son aleatorios.

Atributos de un array

Existen varios atributos y funciones que describen las características de un array.

- a.ndim: Devuelve el número de dimensiones del array a.
- a.shape: Devuelve una tupla con las dimensiones del array a.
- a.size: Devuelve el número de elementos del array a.
- a.dtype: Devuelve el tipo de datos de los elementos del array a.

Acceso a los elementos de un array

Para acceder a los elementos contenidos en un array se usan índices al igual que para acceder a los elementos de una lista, pero indicando los índices de cada dimensión separados por comas.
Al igual que para listas, los índices de cada dimensión comienzan en 0.

También es posible obtener subarrays con el operador dos puntos : indicando el índice inicial y el siguiente al final para cada dimensión, de nuevo separados por comas.

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a[1, 0]) # Acceso al elemento de la fila 1 columna 0
print(a[1][0]) # Otra forma de acceder al mismo elemento
print(a[:, 0:2])
```

Filtrado de elementos de un array

Una característica muy útil de los arrays es que es muy fácil obtener otro array con los elementos que cumplen una condición.
a[condición]: Devuelve una lista con los elementos del array a que cumplen la condición.

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
print(a[(a % 2 == 0)])
```

Operaciones aritméticas

NumPy facilita realizar operaciones aritméticas con arreglos:

- Se pueden usar el arreglo y un sólo número,
- O realizar las operaciones entre dos arreglos.

Operaciones a nivel de elemento:

Suma (+):

```
import numpy as np
arr_suma_elemento = np.array([1, 2, 3, 4, 5]) + 10
print(arr_suma_elemento)
```

Resta (-):

```
import numpy as np
arr_resta_elemento = np.array([1, 2, 3, 4, 5]) - 10
print(arr_resta_elemento)
```

Multiplicación (*):

```
import numpy as np
arr_multiplicacion_elemento = np.array([1, 2, 3, 4, 5]) * 2
print(arr_multiplicacion_elemento)
```

División (/):

```
import numpy as np
arr_division_elemento = np.array([1, 2, 3, 4, 5]) / 2
print(arr_division_elemento)
```

Módulo (%):

```
import numpy as np
arr_modulo_elemento = np.array([10, 11, 12, 13, 14]) % 3
print(arr_modulo_elemento)
```

Potencia (**):

```
import numpy as np
arr_potencia_elemento = np.array([1, 2, 3, 4, 5]) ** 2
print(arr_potencia_elemento)
```

Operaciones entre matrices:

Suma (+):

```
import numpy as np
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([6, 7, 8, 9, 10])
arr_suma_arreglos = arr1 + arr2
print(arr_suma_arreglos)
```

Resta (-):

```
import numpy as np
arr_resta_arreglos = arr2 - arr1
print(arr_resta_arreglos)
```

Multiplicación (*):

```
import numpy as np
arr_multiplicacion_arreglos = arr1 * arr2
print(arr_multiplicacion_arreglos)
```

División (/):

```
import numpy as np
arr_division_arreglos = arr2 / arr1
print(arr
```

Tema 6: Ciencia de Datos

¿Qué es ciencia de datos?

Introducción

La ciencia de datos es un campo interdisciplinario que utiliza métodos, procesos, algoritmos y sistemas científicos para extraer conocimiento y conclusiones de datos estructurados y no estructurados. La ciencia de datos combina técnicas de varias disciplinas como matemáticas, estadística, informática y conocimiento de negocio para analizar y entender fenómenos reales mediante el uso de datos.

¿Por qué es importante la ciencia de datos?

En el mundo actual, generamos una cantidad enorme de datos cada día. Desde las redes sociales, transacciones comerciales, sensores de dispositivos IoT, hasta registros médicos y sistemas educativos. La ciencia de datos permite transformar estos datos en información valiosa para tomar decisiones informadas, predecir tendencias futuras, y automatizar procesos.

Un Ejemplo Simple

Imagina que trabajas en una tienda en línea y quieres aumentar las ventas. Tienes acceso a una gran cantidad de datos, como las compras anteriores de los clientes, las visitas a tu página web, los comentarios en redes sociales, etc. Como científico de datos, podrías hacer lo siguiente:

1. **Recolección de Datos**: Recopilar datos de diversas fuentes como registros de ventas, comportamiento de navegación en el sitio web, y encuestas de satisfacción de clientes.
2. **Limpieza de Datos**: Asegurarte de que los datos sean precisos y estén completos. Por ejemplo, manejando valores faltantes o eliminando datos duplicados.
3. **Análisis de Datos**: Usar técnicas estadísticas para encontrar patrones en los datos. Por ejemplo, podrías descubrir que los clientes que compran productos electrónicos tienden a comprar accesorios una semana después.
4. **Modelado Predictivo**: Aplicar algoritmos de aprendizaje automático para predecir comportamientos futuros. Por ejemplo, podrías crear un modelo que prediga qué productos es más probable que compren ciertos clientes.
5. **Visualización de Datos**: Crear gráficos y dashboards que muestren tus hallazgos de manera clara y comprensible para los tomadores de decisiones.
6. **Toma de Decisiones**: Usar la información derivada de los datos para implementar estrategias de marketing dirigidas, mejorar la experiencia del cliente, y optimizar el inventario.

Conceptos Clave

Datos

Los datos son hechos y estadísticas recopilados juntos para referencia o análisis. Pueden estar en diversas formas, tales como:

- **Estructurados**: Datos organizados en formatos tabulares como bases de datos relacionales.
- **No estructurados**: Datos no organizados en un formato predefinido, como textos, imágenes, videos, etc.

Información

La información es el procesamiento, interpretación y organización de datos de manera que tengan significado o sean útiles para tomar decisiones.

Conocimiento

El conocimiento es la aplicación de información y datos para tomar decisiones y resolver problemas. Se construye a partir de la información mediante la experiencia, contexto, interpretación y reflexión.

Fundamentos de la Ciencia de Datos

Estadísticas

Las estadísticas son fundamentales en la ciencia de datos para la recolección, análisis, interpretación y presentación de datos. Permiten a los científicos de datos entender mejor los datos y derivar conclusiones válidas.

Aprendizaje Automático (Machine Learning)

El aprendizaje automático es una rama de la inteligencia artificial que permite a las máquinas aprender de los datos sin ser explícitamente programadas. Utiliza algoritmos que iteran sobre los datos para encontrar patrones y mejorar las predicciones con el tiempo.

Análisis de Datos

El análisis de datos implica inspeccionar, limpiar y modelar datos con el objetivo de descubrir información útil, concluir y apoyar la toma de decisiones.

Big Data

El big data se refiere a conjuntos de datos que son tan grandes o complejos que las aplicaciones tradicionales de procesamiento de datos no son adecuadas. Los retos incluyen la captura de datos, almacenamiento, análisis, búsqueda, compartir, transferencia, visualización y privacidad de la información.

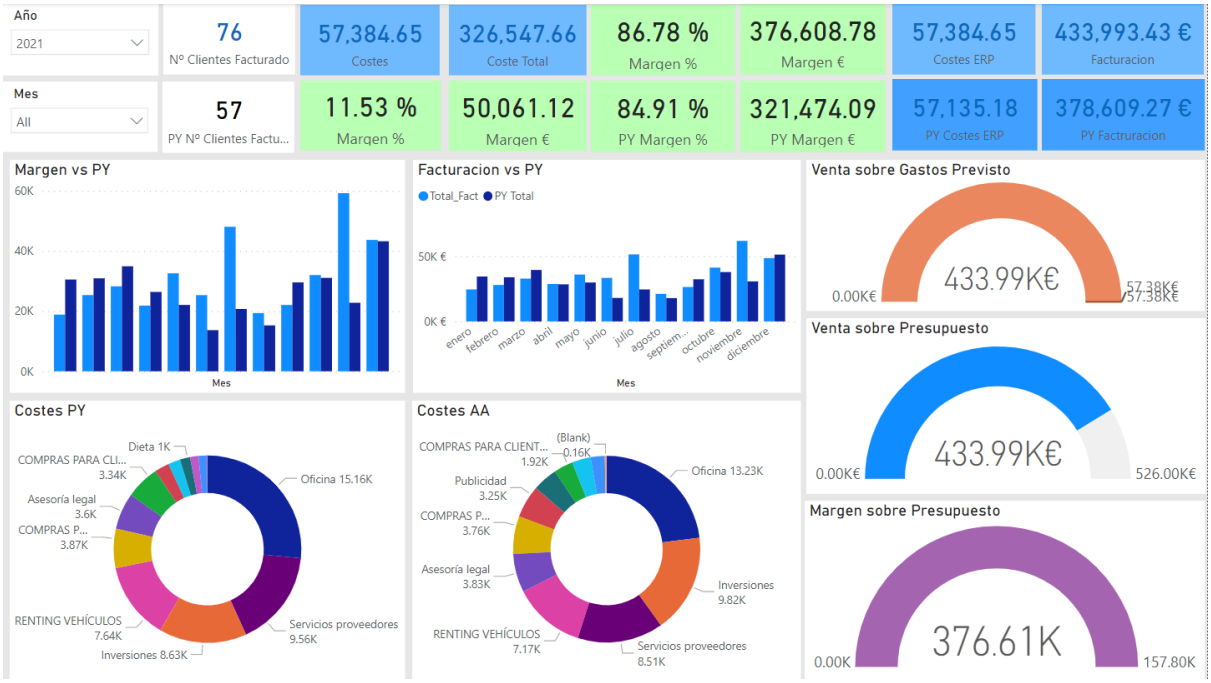
Herramientas Comunes en Ciencia de Datos

- **Python**: Un lenguaje de programación muy popular en la ciencia de datos debido a su simplicidad y la gran cantidad de bibliotecas disponibles (pandas, NumPy, scikit-learn, etc.).
- **SQL**: Lenguaje de consulta estructurado utilizado para gestionar y manipular bases de datos relacionales.
- **Hadoop**: Un marco de software que soporta el procesamiento distribuido de grandes conjuntos de datos.

Importancia de la Ciencia de Datos

La ciencia de datos es crucial en el mundo actual porque permite a las organizaciones:

- **Tomar decisiones informadas**: Basadas en datos y análisis rigurosos.
- **Identificar tendencias y patrones**: Que pueden ser utilizados para ganar ventajas competitivas.
- **Automatizar procesos**: Mejorando la eficiencia y reduciendo costos.
- **Mejorar productos y servicios**: A través de la personalización y la predicción de necesidades.



Conclusión

La ciencia de datos es una disciplina poderosa que combina múltiples campos del conocimiento para extraer información valiosa de los datos. En esta asignatura, exploraremos los fundamentos de la ciencia de datos y aprenderemos a utilizar Python para analizar y visualizar datos de manera efectiva.

Tema 7 : ¿Qué es Python?

¿Qué es Python?

Introducción

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, que se destaca por su simplicidad y legibilidad. Fue creado por Guido van Rossum y su primera versión fue lanzada en 1991. Python ha crecido en popularidad debido a su facilidad de aprendizaje y su vasta colección de bibliotecas y frameworks que facilitan el desarrollo de aplicaciones en diversos campos, incluyendo la ciencia de datos.

Historia de Python

Python fue concebido a finales de los años 80 y su implementación comenzó en diciembre de 1989. Guido van Rossum, el creador de Python, tenía como objetivo crear un lenguaje que combinara el poder de los lenguajes de alto nivel con la facilidad de lectura de los lenguajes de scripting. La primera versión oficial, Python 1.0, fue lanzada en 1991.

Hitos Importantes

- **1991**: Lanzamiento de Python 1.0.
- **2000**: Lanzamiento de Python 2.0, introduciendo nuevas características como el recolector de basura y el soporte para Unicode.
- **2008**: Lanzamiento de Python 3.0, una versión no retrocompatible que solucionó muchos problemas y limitaciones de las versiones anteriores.
- **2020**: Fin del soporte oficial para Python 2.x, marcando una transición completa a Python 3.x.

Características Principales de Python

Simplicidad y Legibilidad

Python se diseñó para ser fácil de leer y escribir. Su sintaxis clara y su estructura basada en la indentación permiten a los programadores expresar conceptos en menos líneas de código en comparación con otros lenguajes de programación.

Gran Biblioteca Estándar

Python incluye una extensa biblioteca estándar que proporciona módulos y funciones para realizar tareas comunes, como manipulación de archivos, operaciones matemáticas, acceso a sistemas operativos y desarrollo de interfaces gráficas.

Multiparadigma

Python admite múltiples paradigmas de programación, incluyendo la programación procedimental, orientada a objetos y funcional. Esta flexibilidad permite a los desarrolladores elegir el estilo de programación que mejor se adapte a sus necesidades.

Interpretado

Python es un lenguaje interpretado, lo que significa que el código se ejecuta línea por línea, facilitando la depuración y el desarrollo interactivo.

Comunidad Activa

Python tiene una comunidad activa y creciente que contribuye al desarrollo del lenguaje y crea una gran cantidad de bibliotecas y herramientas de terceros.

Python en Ciencia de Datos

Bibliotecas Específicas

Python se ha convertido en el lenguaje preferido para la ciencia de datos debido a sus poderosas bibliotecas específicas, como:

- **NumPy**: Proporciona soporte para arrays y matrices multidimensionales, junto con una colección de funciones matemáticas para operar con ellos.
- **pandas**: Facilita la manipulación y análisis de datos, proporcionando estructuras de datos y funciones para trabajar con datos etiquetados y heterogéneos.
- **Matplotlib**: Utilizada para crear visualizaciones estáticas, animadas e interactivas en Python.
- **scikit-learn**: Proporciona herramientas simples y eficientes para el análisis y modelado de datos, incluyendo algoritmos de aprendizaje automático.
- **TensorFlow y PyTorch**: Bibliotecas para el desarrollo y entrenamiento de modelos de aprendizaje profundo.

Facilidad de Integración

Python se integra fácilmente con otros lenguajes y tecnologías. Por ejemplo, puede interactuar con código escrito en C, C++, Java y otros lenguajes. También es compatible con bases de datos SQL y NoSQL, y puede comunicarse con servicios web a través de API RESTful.

Amplia Comunidad y Recursos

La popularidad de Python en la ciencia de datos ha llevado a la creación de una amplia variedad de recursos educativos, como tutoriales, cursos en línea, libros y foros de discusión. Esto facilita el aprendizaje y el desarrollo profesional en este campo.

Ejemplo Simple

Aquí hay un ejemplo simple de cómo Python se puede utilizar para analizar un conjunto de datos:

```
import pandas as pd

import matplotlib.pyplot as plt

# Cargar un conjunto de datos desde un archivo CSV

data = pd.read_csv('ventas.csv')

# Mostrar las primeras filas del conjunto de datos

print(data.head())

# Graficar las ventas mensuales

data.plot(x='Mes', y='Ventas', kind='bar')

plt.title('Ventas Mensuales')

plt.xlabel('Mes')

plt.ylabel('Ventas')

plt.show()
```

Tema 8: Sintaxis semántica y reglas de python

Sintaxis, Semántica y Reglas de Python

Introducción

Para dominar cualquier lenguaje de programación, es fundamental entender su sintaxis y semántica. La sintaxis se refiere a las reglas que dictan cómo se debe escribir el código, mientras que la semántica se ocupa del significado de las construcciones del lenguaje. Python es conocido por su sintaxis clara y legible, lo que lo convierte en una excelente opción tanto para principiantes como para programadores experimentados.

Sintaxis de Python

Indentación

A diferencia de muchos otros lenguajes de programación, Python utiliza la indentación para definir bloques de código. Esto hace que el código sea más legible y estructurado.

Ejemplo:

```
if True:
    print("Esto está dentro del bloque if")
    if False:
        print("Esto no se imprimirá")
print("Esto está fuera del bloque if")
```

Comentarios

Los comentarios en Python se utilizan para explicar el código y no se ejecutan. Se pueden hacer comentarios de una sola línea usando # o comentarios de varias líneas utilizando comillas triples """.

Ejemplo:

```
# Esto es un comentario de una sola línea
"""
Esto es un comentario
de varias líneas
"""
print("Hola, Mundo!")  # Esto imprime un mensaje
```

Variables y Tipos de Datos

Las variables en Python se crean automáticamente cuando se asigna un valor a un identificador. No es necesario declarar el tipo de variable explícitamente.

Ejemplo:

```
x = 10          # Entero
y = 3.14        # Flotante
```

```
nombre = "Ana"  # Cadena de texto

es_activo = True  # Booleano
```

Operadores Básicos

Python soporta una variedad de operadores, como operadores aritméticos, de comparación, lógicos y de asignación.

Ejemplo:

```
# Operadores aritméticos

a = 10 + 5

b = 10 - 5

c = 10 * 5

d = 10 / 5

e = 10 % 3


# Operadores de comparación

f = (10 == 5)   # False

g = (10 != 5)   # True

h = (10 > 5)    # True


# Operadores lógicos

i = (True and False)  # False
```

```
j = (True or False)   # True

# Operadores de asignación

k = 10

k += 5  # k = k + 5
```

Semántica de Python

Tipado Dinámico

Python es un lenguaje de tipado dinámico, lo que significa que el tipo de una variable se determina en tiempo de ejecución y no es necesario declararlo explícitamente. Esto hace que Python sea flexible pero también puede introducir errores si no se tiene cuidado.

Ejemplo:

```
x = 10        # x es un entero

x = "diez"    # ahora x es una cadena de texto
```

Mutabilidad

En Python, algunos tipos de datos son mutables (pueden ser cambiados después de su creación) mientras que otros son inmutables (no pueden ser cambiados después de su creación).

Ejemplos:

```
# Lista (mutable)

lista = [1, 2, 3]

lista.append(4)  # lista ahora es [1, 2, 3, 4]


# Tupla (inmutable)
```

```
tupla = (1, 2, 3)

# tupla.append(4)  # Esto dará un error
```

Manejo de Errores

Python utiliza estructuras de control de flujo para manejar errores y excepciones. La construcción try...except se usa para capturar y manejar excepciones.

Ejemplo:

```
try:

    division = 10 / 0

except ZeroDivisionError:

    print("Error: División por cero")
```

Reglas y Buenas Prácticas en Python

Nombres de Variables y Funciones

- Utiliza nombres descriptivos y en minúsculas, separando las palabras con guiones bajos (_).
- Las funciones también deben tener nombres descriptivos y seguir la misma convención.

Ejemplo:

```
mi_variable = 5

def suma_dos_numeros(a, b):

    return a + b
```

Estructura de Código

- Mantén líneas de código de menos de 80 caracteres.
- Utiliza espacios en lugar de tabulaciones para la indentación (4 espacios por nivel de indentación).

Documentación

- Utiliza docstrings para documentar módulos, clases y funciones.
- Incluye comentarios solo cuando sea necesario para entender el código.

Ejemplo:

```
def area_circulo(radio):
    """
    Calcula el área de un círculo dado su radio.
    Parámetros:
    radio (float): El radio del círculo.
    Retorna:
    float: El área del círculo.
    """
    return 3.1416 * radio ** 2
```

Conclusión

Comprender la sintaxis, la semántica y las reglas de Python es crucial para escribir código efectivo y mantenible. Python se destaca por su simplicidad y claridad, lo que permite a los desarrolladores concentrarse en resolver problemas complejos sin preocuparse por detalles sintácticos complicados.

UNIDAD 2

Tema 1: Pandas

Introducción a la librería Pandas para Ciencias de Datos

¿Qué es Pandas?

Pandas es una biblioteca de Python especializada en el análisis de datos y la manipulación de estructuras de datos tabulares. Fue creada originalmente por Wes McKinney en 2008 y desde entonces se ha convertido en una de las herramientas más populares y poderosas para el análisis de datos en Python. Una de sus principales características es el uso de los **DataFrames y Series**.

Un DataFrame es una estructura bidimensional similar a una tabla.
Una Serie es una estructura unidimensional similar a un array.

Instalación de Pandas

Pandas es una biblioteca de Python, así que puedes instalarla usando pip, el gestor de paquetes de Python:

Copiar código

```
pip install pandas
```

Importar Pandas

Una vez que tienes Pandas instalado, puedes importarlo en tu script de Python:

javascript

Copiar código

```
import pandas as pd
```

Creación de un DataFrame

El DataFrame es una estructura de datos fundamental en Pandas. Puedes crear uno a partir de listas, diccionarios, arreglos de numpy, etc.

Crear un DataFrame a partir de una lista de listas:

```
data = [    ['Alice', 25],

    ['Bob', 30],

    ['Charlie', 35]

]

df = pd.DataFrame(data, columns=['Nombre', 'Edad'])
```

Crear un DataFrame a partir de un diccionario:

```
data = {

    'Nombre': ['Alice', 'Bob', 'Charlie'],

    'Edad': [25, 30, 35]

}

df = pd.DataFrame(data)
```

Leer y escribir datos

Pandas puede leer y escribir datos desde y hacia diferentes formatos como CSV, Excel, bases de datos SQL, etc.

```
# Leer un archivo CSV

df = pd.read_csv('datos.csv')

# Escribir a un archivo CSV

df.to_csv('nuevo_datos.csv', index=False)

# Leer un archivo Excel

df = pd.read_excel('datos.xlsx')

# Escribir a un archivo Excel

df.to_excel('nuevo_datos.xlsx', index=False)
```

Explorar un DataFrame

Pandas ofrece varias funciones para explorar y entender tus datos.

```
# Mostrar las primeras filas del DataFrame

print(df.head())

# Obtener información sobre el DataFrame

print(df.info())

# Resumen estadístico del DataFrame

print(df.describe())

# Seleccionar una columna

print(df['Nombre'])

# Seleccionar varias columnas

print(df[['Nombre', 'Edad']])
```

Manipulación de datos

Pandas proporciona numerosas funciones para manipular y transformar datos.

```
# Filtrar filas basadas en una condición

personas_jovenes = df[df['Edad'] < 30]

# Agregar una nueva columna

df['Género'] = ['F', 'M', 'M']

# Eliminar una columna

df.drop('Género', axis=1, inplace=True)

# Renombrar una columna

df.rename(columns={'Nombre': 'Nombre completo'}, inplace=True)
```

Operaciones con datos faltantes

Pandas facilita el manejo de valores faltantes en los datos.

```
# Eliminar filas con valores faltantes

df.dropna(inplace=True)

# Llenar valores faltantes con un valor específico

df.fillna(0, inplace=True)

# Llenar valores faltantes con la media de la columna

df.fillna(df.mean(), inplace=True)
```

Agrupar y resumir datos

Pandas permite agrupar datos y realizar operaciones de resumen.

```
# Agrupar por una columna y calcular la media de otra columna

datos_agrupados = df.groupby('Categoría')['Valor'].mean()

# Agregar una columna con el recuento de filas por grupo

df['Conteo'] = df.groupby('Categoría')['Categoría'].transform('count')
```

Funciones estadísticas comunes

Pandas ofrece numerosas funciones para realizar cálculos estadísticos en los datos.

```
# Calcular la media de una columna

print(df['Edad'].mean())

# Calcular la mediana de una columna

print(df['Edad'].median())

# Calcular la desviación estándar de una columna

print(df['Edad'].std())

# Calcular el máximo de una columna

print(df['Edad'].max())

# Calcular el mínimo de una columna

print(df['Edad'].min())
```

```
# Calcular la correlación entre dos columnas

print(df['Columna1'].corr(df['Columna2']))
```

Tema 2: Matplotib

Introducción a Matplotlib para gráficos estadísticos

Matplotlib es el “abuelo” de las librerías de visualización de datos con Python. Fue creado por John Hunter. Lo creó para tratar de replicar las capacidades de graficar de MatLab en Python. Es una excelente biblioteca de gráficos 2D y 3D para generar figuras científicas.

Algunos de los principales Pros de Matplotlib son:

- Generalmente es fácil comenzar por gráficas simples.
- Soporte para etiquetas personalizadas y textos.
- Gran control de cada elemento en una figura.
- Salida de alta calidad en muchos formatos.
- Muy personalizable en general.

Matplotlib nos permite crear figuras reproducibles mediante programación. La página web oficial de Matplotlib: http://matplotlib.org/

Instalación

Se puede instalar con el siguiente comando:

```
pip install matplotlib
```

Importar la librería

Importar el módulo matplotlib.pyplot con el nombre de plt (esto es un estándar en la comunidad):

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

Comandos Básicos de Matplotlib

Veamos un ejemplo muy simple usando dos arreglos numpy. También se pueden usar listas, pero lo más probable es usar arreglos Numpy o columnas de pandas (que esencialmente también se comportan como arreglos). Los datos que queremos graficar:

```
import numpy as np

x = np.linspace(0, 5, 11)

y = x ** 2
```

```
# Método básico para graficar X vs Y

plt.plot(x, y) # se grafica una línea de color azul

plt.show() # Mostrar la gráfica luego de que ya se definieron todos los elementos
```

Título

```
plt.plot(x, y) # se grafica una línea de color azul

plt.title('Título de la gráfica')  # Definir el título de la gráfica
```

O también se puede utilizar las columnas de un DataFrame de Pandas:

```
import matplotlib.pyplot as plt

import pandas as pd

df = pd.DataFrame({ "x": [1, 2, 3, 4, 5], "y": [6, 7, 8, 9, 0]})

plt.plot(df["x"], df["y"])

plt.title("Título con pandas")

plt.show()
```

Nombres de los ejes

```
plt.plot(x, y)  # se grafica una línea de color azul

plt.xlabel('Nombre del eje X')  # Definir el nombre del eje X

plt.ylabel('Nombre del eje Y')  # Definir el nombre del eje Y

plt.title('Título de la gráfica')  # Definir el título de la gráfica
```

Leyenda

Puede usar el argumento de palabra clave label = “texto de etiqueta” cuando se agreguen gráficas u otros objetos a la figura, y luego usar el método legend sin argumentos para agregar la leyenda a la figura:

```
plt.plot(x, y, label="x vs y")  # se grafica una línea de color azul

# Se pone en el atributo 'label' el texto deseado

plt.xlabel('Nombre del eje X')  # Definir el nombre del eje X

plt.ylabel('Nombre del eje Y')  # Definir el nombre del eje Y

plt.title('Título de la gráfica')  # Definir el título de la gráfica

plt.legend()  # Agregar la leyenda al plot
```

¡La leyenda se superpone con parte de la gráfica!

El método legend toma un argumento opcional de palabra clave loc que puede usarse para especificar en qué parte de la figura debe dibujarse la leyenda. Los valores permitidos de loc son códigos numéricos para los diversos lugares donde se puede dibujar la leyenda.

Cuadrícula (Grid)

```
plt.plot(x, y, label="x vs y")  # Se grafica una línea de color azul

# Se pone en el atributo 'label' el texto deseado

plt.xlabel('Nombre del eje X')  # Definir el nombre del eje X

plt.ylabel('Nombre del eje Y')  # Definir el nombre del eje Y

plt.title('Título de la gráfica')  # Definir el título de la gráfica

plt.legend()  # Agregar el legend al plot

plt.grid(True)  # Poner grid en la gráfica
```

Parámetros de las líneas: colores, ancho y tipos

Matplotlib le brinda muchas opciones para personalizar colores, anchos de línea y tipos de línea.

Colores Básicos

Con matplotlib, podemos definir los colores de las líneas y otros elementos gráficos de varias maneras. En primer lugar, podemos usar la sintaxis similar a MATLAB donde 'b' significa azul, 'g' significa verde, etc. También se admite la API MATLAB para seleccionar estilos de línea: donde, por ejemplo, ‘b.-‘ significa una línea azul con puntos:

```
# Estilo MATLAB de estilo y color de línea

plt.plot(x, x**2, 'b.-')  # Línea azul con puntos

plt.plot(x, x**3, 'g--')  # Línea verde discontinua
```

Colores usando el parámetro color

También podemos definir colores por sus nombres o códigos hexadecimales RGB y, opcionalmente, proporcionar un valor alpha utilizando los argumentos de las palabras clave color y alpha. Alpha indica opacidad.

```
plt.plot(x, x, color="red")  # Rojo

plt.plot(x, x+1, color="red", alpha=0.5)  # Medio transparente

plt.plot(x, x+2, color="#8B008B")  # RGB hex code

plt.plot(x, x+3, color="#F08C08")  # RGB hex code

plt.grid(True)  # Poner grid en la gráfica
```

Estilos de líneas y marcadores

Para cambiar el ancho de línea, podemos usar el argumento de la palabra clave linewidth o lw. El estilo de línea se puede seleccionar usando los argumentos de palabras clave linestyle o ls:

```
plt.subplots(figsize=(12, 6))

plt.plot(x, x+1, color="red", linewidth=0.25)

plt.plot(x, x+2, color="red", linewidth=0.50)
```

```
plt.plot(x, x+3, color="red", linewidth=1.00)

plt.plot(x, x+4, color="red", linewidth=2.00)

# Posibles opciones linestype '-', '-', '-.', ':', 'steps'

plt.plot(x, x+5, color="green", lw=3, linestyle='-')

plt.plot(x, x+6, color="green", lw=3, ls='-.')

plt.plot(x, x+7, color="green", lw=3, ls=':')
```

Anotaciones de texto

Anotar texto en figuras matplotlib se puede hacer usando la función text. Es compatible con el formato LaTeX al igual que los textos y títulos de la etiqueta del eje:

```
# Datos para graficar

xx = np.linspace(-0.75, 1., 100)

plt.plot(xx, xx**2, xx, xx**3)

plt.title("Plot con anotaciones")

# Anotación 1

plt.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")

# Anotación 2

plt.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green")
```

Tipos Especiales de Plots

Scatter Plot (Dispersión)

```
# Gráfica X vs Y

# Crear datos aleatorios

N = 50

x = np.random.rand(N)

y = np.random.rand(N)
```

```
plt.scatter(x, y)

plt.title("Scatter plot Simple")

plt.show()  # En jupyter notebook no es necesario este comando
```

Con las gráficas de dispersión o scatter se pueden representar más de 2 variables en una misma gráfica. En el siguiente ejemplo, se realizará la comparación de x vs y, el color de los puntos se representará con otra variable y el tamaño de los puntos será otra variable.

```
# Se creará otra variable que se representará con colores

colors = np.random.rand(N)  # Usar colores aleatorios

# Se creará otra variable que se representará con el área de los puntos

area = np.pi * (15 * np.random.rand(N))**2  # 0 to 15 punto radio

plt.scatter(x, y, s=area, c=colors, alpha=0.5)  # El atributo alpha es para
la transparencia

plt.title("Scatter plot de representación de 4 variables")
```

Histograma

Un histograma es una representación gráfica de una variable en forma de barras, donde la superficie de cada barra es proporcional a la frecuencia de los valores representados.

```
# Crear datos aleatorios

from random import sample

data = sample(range(1, 1000), 100)

plt.hist(data, bins=10)  # bins el número de divisiones del histograma

plt.title("Histograma")
```

Diagramas de torta

```
labels = 'Caballos', 'Cerdos', 'Perros', 'Vacas'

sizes = [15, 30, 45, 10]

explode = (0, 0.1, 0, 0)  # Solo "Sacar" (explotar) la segunda sección

plt.pie(sizes, explode=explode, labels=labels, autopct='%1.1f%%',
shadow=True, startangle=140)
```

```
plt.title("Gráfica Pie")

plt.show()
```

Gráfico de barras

```
objects = ('Manzana', 'Banana', 'Cereza', 'Durazno')

y_pos = np.arange(len(objects))

performance = [10, 20, 30, 40]

plt.bar(y_pos, performance, align='center')

plt.xticks(y_pos, objects)

plt.ylabel('Cantidades')

plt.title('Gráfico de Barras')

plt.show()
```

Conclusiones

Recuerda que Matplotlib es una biblioteca extremadamente flexible, pero tiene una curva de aprendizaje moderada, ya que requiere manejar un alto nivel de detalle para crear visualizaciones más complejas.

Tema 3: Limpieza y estructura de datos

Limpieza y estructura de los datos

La limpieza de datos es una tarea crucial en el proceso de analítica avanzada. Si los datos no están limpios, los resultados del análisis pueden ser inexactos y llevar a decisiones equivocadas. Python es uno de los lenguajes de programación más utilizados en el mundo de la analítica avanzada y la ciencia de datos, por lo que es una herramienta poderosa para la limpieza de datos.

Pasos para la limpieza de datos:

Identificar y eliminar valores atípicos o valores faltantes:
Este paso implica la identificación de valores atípicos o missing values (valores faltantes) en el conjunto de datos. Los valores atípicos son observaciones inusuales que difieren significativamente del resto de los datos, mientras que los valores faltantes son aquellos para los cuales no se registra ningún valor. La eliminación de estos valores ayuda a garantizar la precisión y la integridad de los datos.

Eliminar datos duplicados:
En esta etapa, se identifican y eliminan registros duplicados en el conjunto de datos. La presencia de datos duplicados puede distorsionar el análisis y conducir a conclusiones incorrectas. Al eliminar estos duplicados, se garantiza que cada observación sea única y que no haya redundancia de información.

Convertir los datos en el formato adecuado:
A veces, los datos pueden estar en formatos incorrectos o inconsistentes. En este paso, se realiza la conversión de los datos al formato adecuado según las necesidades del análisis. Esto puede incluir la conversión de tipos de datos (por ejemplo, de texto a numérico) o la estandarización de formatos (por ejemplo, fechas en diferentes formatos).

Normalizar los datos:
La normalización de datos implica escalar los valores de los datos a un rango específico, lo que facilita la comparación y el análisis. Esto es especialmente importante cuando se trabaja con variables que tienen diferentes unidades o escalas. La normalización asegura que todas las variables contribuyan de manera equitativa al análisis.

Verificar la consistencia de los datos:
En esta etapa, se verifica la consistencia de los datos para identificar posibles errores o inconsistencias. Esto puede incluir la validación de rangos de valores, la coherencia entre variables relacionadas y la identificación de datos que contradicen las reglas establecidas.

Verificar la validez de los datos:
Finalmente, se verifica la validez de los datos para asegurarse de que sean precisos y representen con precisión el fenómeno o proceso que se está estudiando. Esto puede implicar la comparación de los datos con fuentes externas confiables o la validación cruzada con otros conjuntos de datos.

Tema 4: Limpieza y estructura de datos con Pandas

Limpieza y Estructura de los Datos con Pandas

Introducción

En el análisis de datos, la limpieza y estructuración de los datos son pasos críticos para garantizar la precisión y la calidad de los resultados. Los datos crudos a menudo contienen errores, valores faltantes y formatos inconsistentes que deben ser abordados antes de realizar cualquier análisis. En esta clase, aprenderemos técnicas fundamentales para la limpieza y estructuración de datos utilizando Python.

Importancia de la Limpieza de Datos

La limpieza de datos es esencial por varias razones:

- **Precisión:** Los datos sucios pueden llevar a conclusiones incorrectas y análisis poco fiables.
- **Consistencia:** Garantiza que los datos estén en un formato uniforme, facilitando su análisis.
- **Eficiencia:** Mejora el rendimiento de los algoritmos de análisis y machine learning.

Herramientas para la Limpieza de Datos en Python

Python ofrece varias bibliotecas poderosas para la manipulación y limpieza de datos, incluyendo:

Pandas

Pandas es una biblioteca fundamental para la manipulación de datos en Python. Proporciona estructuras de datos como `DataFrame` y `Series`, que son extremadamente útiles para la limpieza y análisis de datos.

Instalación

```
pip install pandas
```

Importación

```
import pandas as pd
```

Ejemplos de limpieza de datos

Cargar Datos

Para empezar, carguemos un conjunto de datos en un DataFrame de Pandas.

```
df = pd.read_csv('data.csv')
```

Inspeccionar Datos

Es importante revisar los datos para entender su estructura y contenido.

```
print(df.head())  # Muestra las primeras 5 filas del DataFrame

print(df.info())  # Muestra un resumen de la información del DataFrame

print(df.describe())  # Muestra estadísticas descriptivas
```

Manejo de valores faltantes

Los valores faltantes son comunes en los conjuntos de datos y deben ser manejados adecuadamente.

Identificar Valores Faltantes

```
print(df.isnull().sum())  # Muestra el número de valores nulos por columna
```

Eliminar Filas o Columnas con Valores Faltantes

```
df = df.dropna()  # Elimina todas las filas con al menos un valor nulo

df = df.dropna(axis=1)  # Elimina todas las columnas con al menos un valor nulo
```

Rellenar Valores Faltantes

```
df = df.fillna(0)  # Rellena valores nulos con 0

df = df.fillna(df.mean())  # Rellena valores nulos con la media de cada columna
```

Eliminación de Duplicados

Los datos duplicados pueden distorsionar los análisis y deben ser eliminados.

```
df = df.drop_duplicates()  # Elimina filas duplicadas
```

Conversión de Tipos de Datos

Asegurarse de que cada columna tenga el tipo de dato correcto es crucial para el análisis.

```
df['columna'] = df['columna'].astype(float)  # Convierte una columna a tipo float

df['fecha'] = pd.to_datetime(df['fecha'])  # Convierte una columna a tipo datetime
```

Manejo de Datos Inconsistentes

A menudo, los datos pueden tener formatos inconsistentes que deben ser unificados.

```
df['columna'] = df['columna'].str.strip()  # Elimina espacios en blanco al inicio y al final de una cadena

df['columna'] = df['columna'].str.lower()  # Convierte todas las cadenas a minúsculas
```

Conclusión

La limpieza y estructuración de datos es una parte esencial del proceso de análisis de datos. Utilizando herramientas como Pandas en Python, podemos abordar problemas comunes en los datos crudos, como valores faltantes, duplicados y tipos de datos incorrectos. Una vez que los datos están limpios y estructurados, podemos proceder con confianza a la etapa de análisis.

Tema 5: Obtención de datos a partir de base de datos

Obtención de Datos a partir de Bases de Datos SQL

La capacidad de interactuar con bases de datos SQL desde Python es una habilidad fundamental para cualquier desarrollador o analista de datos. Python ofrece varias bibliotecas que facilitan la conexión y la manipulación de datos en bases de datos SQL, como MySQL, PostgreSQL y MariaDB. Estas bibliotecas proporcionan funciones y métodos que permiten establecer conexiones, ejecutar consultas SQL y procesar resultados de manera eficiente.

Instalación del paquete para conectar con MySQL

```
pip install mysqlclient
```

Conexión a la Base de Datos

Para conectar a la base de datos, utilizamos el método `connect` de la librería `MySQLdb`. Los parámetros que debemos especificar son:

- La dirección del servidor de base de datos
- El usuario que vamos a usar para la conexión
- La contraseña del usuario
- La base de datos a la que nos vamos a conectar

Esto nos devuelve un objeto que representa la conexión. Es importante manejar las excepciones para capturar errores durante la conexión y cerrar la conexión con `close()` una vez finalizado el trabajo.

```
1. import sys
2. import MySQLdb
3.
4. try:
5.     db = MySQLdb.connect("localhost", "root", "", "testdb")
6. except MySQLdb.Error as e:
7.     print("No se pudo conectar a la base de datos:", e)
8.     sys.exit(1)
9.
10.   print("Conexión correcta.")
11.   db.close()
```

Ejecución de Instrucciones SQL

Para ejecutar instrucciones SQL, utilizamos un objeto de la clase `cursor`. El siguiente ejemplo muestra cómo crear una tabla e insertar datos:

```
1. import sys
2. import MySQLdb
3.
4. try:
5.     db = MySQLdb.connect("localhost", "root", "", "testdb")
6. except MySQLdb.Error as e:
7.     print("No puedo conectar a la base de datos:", e)
8.     sys.exit(1)
9.
10.   cursor = db.cursor()
11.
12.   # Borrar la tabla si existe
13.   cursor.execute("DROP TABLE IF EXISTS empleados")
14.   print("Tabla 'empleados' eliminada (si existía).")
15.
16.   # Crear la tabla
17.   cursor.execute("""
```

```
CREATE TABLE empleados (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50),
    apellido VARCHAR(50),
    edad INT,
    sexo CHAR(1),
    ingreso DECIMAL(10, 2)
)
""")
print("Tabla 'empleados' creada exitosamente.")

# Insertar datos
sql = "INSERT INTO empleados (nombre, apellido, edad, sexo, ingreso) VALUES (%s, %s, %s, %s, %s)"
values = ('José', 'Domingo', 20, 'M', 2000)

try:
    cursor.execute(sql, values)
    db.commit()
    print("Insertado correctamente")
except Exception as e:
    print("Error al insertar los datos", e)
    db.rollback()

db.close()
```

Explicación:

1. **Conexión y Cursor:** Después de realizar la conexión con la base de datos, creamos un objeto `cursor` con `db.cursor()`. Este objeto nos permite ejecutar instrucciones SQL.
2. **Instrucción SQL:** La instrucción que queremos ejecutar se guarda en la variable `sql`. En este caso, estamos insertando un registro en la tabla `empleados`.
3. **Ejecutar Instrucción:** Usamos `cursor.execute(sql, values)` para ejecutar la instrucción, pasando los valores de manera segura para evitar inyecciones SQL.
4. **Confirmar Cambios:** Usamos `db.commit()` para guardar los cambios en la base de datos. Si ocurre un error, usamos `db.rollback()` para revertir los cambios.
5. **Cerrar la Conexión:** Finalmente, cerramos la conexión con `db.close()`.

Lectura de Información de la Tabla

Para leer datos de la base de datos, utilizamos una consulta SQL `SELECT` y luego procesamos los resultados con los métodos del cursor.

Mostrar el Número de Registros Seleccionados:

```
1. import sys
2. import MySQLdb
3.
4. try:
5.     db = MySQLdb.connect("localhost", "root", "", "testdb")
6. except MySQLdb.Error as e:
7.     print("No puedo conectar a la base de datos:", e)
8.     sys.exit(1)
9.
10.   sql = "SELECT * FROM empleados"
11.   cursor = db.cursor()
12.
13.   try:
14.       cursor.execute(sql)
15.       print("Número de registros seleccionados:",
  cursor.rowcount)
16.   except:
17.       print("Error en la consulta")
18.
19.   db.close()
```

Listar los Registros Seleccionados

```
1. import sys
2. import MySQLdb
3.
4. try:
5.     db = MySQLdb.connect("localhost", "root", "", "testdb")
6. except MySQLdb.Error as e:
7.     print("No puedo conectar a la base de datos:", e)
8.     sys.exit(1)
9.
10.   sql = "SELECT * FROM empleados"
11.   cursor = db.cursor()
12.
13.   try:
14.       cursor.execute(sql)
15.       registros = cursor.fetchall()
16.       for registro in registros:
17.           print(registro[0], registro[1], registro[2],
  registro[3], registro[4])
18.   except:
19.       print("Error en la consulta")
20.
```

```
21.   db.close()
```

- **`fetchall()`**: Recupera todos los registros seleccionados. Devuelve una lista de tuplas.
- Podemos recorrer los registros con un ciclo `for`, donde cada registro es una tupla que contiene los valores de cada campo de la fila.

Listar los Registros Usando Diccionario

Si tenemos muchos campos en la tabla, usar índices puede ser incómodo. Para que el cursor devuelva los registros como diccionarios, podemos especificar el tipo de cursor como `DictCursor`.

```
cursor = db.cursor(MySQLdb.cursors.DictCursor)
```

Con esto, cada registro devuelto será un diccionario, y podremos acceder a los campos por su nombre.

python

Copiar código

```
1. import sys
2. import MySQLdb
3.
4. try:
5.     db = MySQLdb.connect("localhost", "root", "", "testdb")
6. except MySQLdb.Error as e:
7.     print("No puedo conectar a la base de datos:", e)
8.     sys.exit(1)
9.
10.sql = "SELECT * FROM empleados"
11.cursor = db.cursor(MySQLdb.cursors.DictCursor)
12.
13.try:
14.    cursor.execute(sql)
15.    registros = cursor.fetchall()
16.    for registro in registros:
17.        print(registro["nombre"])
18.except:
19.    print("Error en la consulta")
20.
21.db.close()
```

Recorriendo los Registros Secuencialmente

Si la tabla contiene muchos registros, utilizar `fetchall()` puede consumir demasiada memoria. En ese caso, podemos usar `fetchone()` para recorrer los registros uno a uno.

python

Copiar código

```
import sys
import MySQLdb

try:
    db = MySQLdb.connect("localhost", "root", "", "testdb")
except MySQLdb.Error as e:
    print("No puedo conectar a la base de datos:", e)
    sys.exit(1)

sql = "SELECT * FROM empleados"
cursor = db.cursor(MySQLdb.cursors.DictCursor)

try:
    cursor.execute(sql)
    registro = cursor.fetchone()
    while registro:
        print(registro["nombre"], registro["apellido"])
        registro = cursor.fetchone()
except:
    print("Error en la consulta")

db.close()
```

Otras Bibliotecas para Conectar a MySQL desde Python

Existen otras bibliotecas disponibles para conectar a MySQL desde Python:

- **mysql-connector-python**: Una biblioteca oficial de Oracle para conectarse a MySQL y MariaDB. Se instala con `pip install mysql-connector-python`.
- **pymysql**: Una biblioteca popular que implementa la API de MySQL en Python. Se instala con `pip install pymysql`.
- **mariadb**: Una biblioteca específica para MariaDB. Se instala con `pip install mariadb`.

Cada una de estas bibliotecas sigue una metodología similar para interactuar con bases de datos SQL en Python.

Tema 6: Obtención de datos a partir de csv

El procesamiento de archivos CSV en Python es una técnica esencial para manipular datos estructurados de manera eficiente. Aquí te explico los conceptos clave y ejemplos para que puedas trabajar con archivos CSV.

1. Abrir archivos con `open()`

El uso de `open()` te permite trabajar con archivos de manera flexible. Existen varios modos para abrir archivos:

- `'r'`: Solo lectura (modo predeterminado).
- `'w'`: Escritura (sobrescribe el archivo si existe).
- `'a'`: Añadir contenido al final del archivo.
- `'rb'`, `'wb'`, `'ab'`: Modos binarios para archivos binarios.

Ejemplo:

```
with open('archivo.csv', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
```

2. Librería `csv`

La librería `csv` proporciona herramientas para leer y escribir archivos CSV de manera eficiente. Aquí los parámetros más importantes que puedes usar con `csv.reader()` y `csv.writer()`:

- `fileobj`: El objeto de archivo abierto (usualmente con `open()`).
- `delimiter`: El carácter que separa los campos en el archivo (por defecto `,`).
- `quotechar`: El carácter utilizado para encerrar campos que contienen delimitadores (por defecto `"`).
- `quoting`: Controla cuándo se deben citar los campos.
- `escapechar`: El carácter para escapar caracteres especiales (opcional).

3. Leer archivos CSV

Para leer un archivo CSV, puedes usar `csv.reader()` que devuelve un iterador sobre las filas del archivo:

```
import csv
with open('localidades.csv', newline='') as archivo_csv:
    lector_csv = csv.reader(archivo_csv, delimiter=',', quotechar='"')
    for fila in lector_csv:
        print(fila)  # Cada fila es una lista de valores
```

4. Escribir archivos CSV

El proceso de escritura se realiza mediante `csv.writer()`:

```
import csv
data = [
    ['Localidad', 'Provincia'],
    ['Buenos Aires', 'Buenos Aires'],
    ['Córdoba', 'Córdoba'],
    ['Rosario', 'Santa Fe']
]
with open('localidades_nuevas.csv', mode='w', newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)  # Escribe múltiples filas
```

5. Acceder a columnas específicas

Si deseas acceder solo a una columna específica, puedes hacerlo fácilmente dentro de un ciclo:

```
# Abrir el archivo CSV en modo lectura
with open('localidades.csv', mode='r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row[0])  # Acceder a la primera columna (localidad)
```

6. Leer como Diccionario

Usando `csv.DictReader()` puedes obtener cada fila como un diccionario, lo que facilita la referencia por nombre de las columnas.

```
# Abrir el archivo CSV en modo lectura
with open('localidades.csv', mode='r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row['localidad'], row['provincia'])  # Acceder por nombre de
columna
```

7. Uso de `with`

El uso de la palabra reservada `with` es fundamental para la gestión eficiente de archivos. Este manejo asegura que el archivo se cierre automáticamente una vez que se haya terminado de trabajar con él, incluso si ocurre un error.

```
with open('localidades.csv', 'r') as archivo:
    # Procesar el archivo
    pass  # El archivo se cierra automáticamente al final
```

Estos ejemplos deberían proporcionarte una base sólida para leer y escribir archivos CSV en Python, lo cual es útil para el análisis de datos y la manipulación de grandes volúmenes de información.

Unidad 3

Tema 1: Algoritmos más utilizados

1. Algoritmos de Búsqueda

1.1 Búsqueda Lineal

La búsqueda lineal es un algoritmo simple que recorre una lista secuencialmente hasta encontrar el elemento buscado.

Código:

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

Complejidad: `O(n)`

1.2 Búsqueda Binaria

La búsqueda binaria es más eficiente que la búsqueda lineal, pero requiere que la lista esté ordenada. El algoritmo divide repetidamente la lista a la mitad hasta encontrar el elemento buscado.

Código:

```
def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
```

```
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return -1
```

Complejidad: `O(log n)`

2. Algoritmos de Ordenamiento

2.1 Ordenamiento por Burbuja

El ordenamiento por burbuja compara cada par de elementos adyacentes y los intercambia si están en el orden incorrecto. Es simple, pero ineficiente para listas grandes.

Código:

```
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

Complejidad: `O(n^2)`

2.2 Ordenamiento Rápido (Quicksort)

Quicksort es un algoritmo eficiente que utiliza el enfoque divide y vencerás. En cada paso, selecciona un pivote y divide la lista en dos partes: una con elementos menores y otra con elementos mayores que el pivote.

Código:

```
def quicksort(lista):
```

```
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[0]
        menores = [x for x in lista[1:] if x <= pivote]
        mayores = [x for x in lista[1:] if x > pivote]
        return quicksort(menores) + [pivote] + quicksort(mayores)
```

Complejidad: `O(n log n)` en promedio

3. Algoritmos de Grafos

3.1 Algoritmo de Dijkstra

Este algoritmo se utiliza para encontrar el camino más corto desde un nodo de inicio a todos los demás nodos en un grafo con pesos no negativos.

Código:

```
import heapq

def dijkstra(grafo, inicio):
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[inicio] = 0
    cola_prioridad = [(0, inicio)]
    while cola_prioridad:
        distancia_actual, nodo_actual =
heapq.heappop(cola_prioridad)
        if distancia_actual > distancias[nodo_actual]:
            continue
        for vecino, peso in grafo[nodo_actual].items():
            distancia = distancia_actual + peso
            if distancia < distancias[vecino]:
                distancias[vecino] = distancia
                heapq.heappush(cola_prioridad, (distancia, vecino))
    return distancias
```

Complejidad: `O((V + E) log V)`

3.2 Algoritmo de Floyd-Warshall

El algoritmo de Floyd-Warshall se utiliza para encontrar las distancias más cortas entre todos los pares de nodos en un grafo.

Código:

```
def floyd_warshall(grafo):
    dist = dict(grafo)
    for k in grafo:
        for i in grafo:
            for j in grafo:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

Complejidad: `O(V^3)`

Conclusión

Estos algoritmos cubren una variedad de necesidades computacionales, desde buscar elementos en una lista, ordenar datos hasta trabajar con grafos para encontrar caminos más cortos. Cada uno de estos algoritmos tiene su campo de aplicación según las características del problema que se esté resolviendo, y el conocimiento de su funcionamiento y complejidad te permitirá elegir la mejor solución para cada escenario.

En próximas clases, exploraremos en mayor detalle algunas optimizaciones y casos de uso más avanzados.

Tema 2: Introducción a la Notación Asintótica

¿Qué es la Notación Asintótica?

La notación asintótica describe el comportamiento de un algoritmo cuando el tamaño de la entrada tiende al infinito, ayudándonos a entender cómo cambia su tiempo de ejecución o uso de memoria en función del tamaño de la entrada. Ignora factores constantes y se enfoca en el crecimiento a largo plazo.

Tipos de Notación Asintótica

Notación Big-O (O)

La notación Big-O describe el **límite superior** del tiempo de ejecución de un algoritmo en el peor caso. Indica el rendimiento más lento que puede tener un algoritmo.

- **Ejemplo:** `f(n) = O(n^2)` para una función `f(n) = 3n^2 + 2n + 1`.

Notación Omega (Ω)

La notación Omega describe el **límite inferior** del tiempo de ejecución de un algoritmo en el mejor caso. Indica el rendimiento más rápido posible que puede tener un algoritmo.

- **Ejemplo:** `f(n) = Ω(n^2)` para una función `f(n) = 3n^2 + 2n + 1`.

Notación Theta (Θ)

La notación Theta describe el **tiempo de ejecución** de un algoritmo cuando está acotado tanto superior como inferiormente. Nos da una idea del comportamiento promedio del algoritmo.

- **Ejemplo:** `f(n) = Θ(n^2)` para una función `f(n) = 3n^2 + 2n + 1`.

Ejemplos Prácticos

1. Búsqueda Lineal

El algoritmo recorre una lista secuencialmente para encontrar un elemento.

Código:

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

-
- **Análisis:**
    - **Peor Caso:** `O(n)`
    - **Mejor Caso:** `Ω(1)`
    - **Caso Promedio:** `Θ(n)`

2. Búsqueda Binaria

La búsqueda binaria divide la lista en mitades de manera repetida para encontrar un elemento, pero solo funciona con listas ordenadas.

Código:

```
def busqueda_binaria(lista, objetivo):
    bajo = 0
```

```
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return -1
```

-
- **Análisis:**
    - **Peor Caso:** `O(log n)`
    - **Mejor Caso:** `Ω(1)`
    - **Caso Promedio:** `θ(log n)`

3. Ordenamiento por Burbuja

El algoritmo de ordenamiento por burbuja compara elementos adyacentes y los intercambia si están en el orden incorrecto.

Código:

```
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

-
- **Análisis:**
    - **Peor Caso:** `O(n^2)`
    - **Mejor Caso:** `Ω(n)` (si la lista ya está ordenada)
    - **Caso Promedio:** `θ(n^2)`

Comparación de Notaciones

Algoritmo	Peor Caso O	Mejor Caso Ω	Caso Promedio Θ
Búsqueda Lineal	`O(n)`	`Ω(1)`	`Θ(n)`
Búsqueda Binaria	`O(log n)`	`Ω(1)`	`Θ(log n)`
Ordenamiento por Burbuja	`O(n^2)`	`Ω(n)`	`Θ(n^2)`

Conclusión

La notación asintótica nos permite describir de forma abstracta la eficiencia de los algoritmos, centrándonos en su comportamiento en el peor, mejor y promedio de los casos. Esto es fundamental para comparar distintos algoritmos y elegir el más adecuado según el problema que se esté resolviendo.

Tema 3: Introducción al Análisis Algorítmico

El **análisis algorítmico** es fundamental para entender la eficiencia y el rendimiento de los algoritmos. Esta clase ha cubierto los conceptos esenciales para evaluar y comparar algoritmos, así como las técnicas comunes que se utilizan en el análisis algorítmico.

¿Qué es el Análisis Algorítmico?

El análisis algorítmico se refiere a la evaluación de la eficiencia de un algoritmo en términos de dos medidas clave:

- **Tiempo de ejecución**: Cuánto tarda un algoritmo en ejecutarse.
- **Uso de memoria**: Cuánta memoria necesita un algoritmo para su ejecución.

Este análisis es importante porque permite:

1. **Evaluar el rendimiento** de los algoritmos.
2. **Prever la escalabilidad** de un algoritmo a medida que aumenta el tamaño de la entrada.
3. **Optimizar** los algoritmos para mejorar su eficiencia.

Medidas de Eficiencia

Tiempo de Ejecución

El tiempo de ejecución de un algoritmo se mide en términos de la cantidad de operaciones que realiza, y se expresa utilizando la notación **Big-O**:

- `O(1)` - **Constante**: El tiempo de ejecución no depende del tamaño de la entrada.
- `O(n)` - **Lineal**: El tiempo de ejecución crece linealmente con el tamaño de la entrada.
- `O(n^2)` - **Cuadrática**: El tiempo de ejecución crece cuadráticamente con el tamaño de la entrada.

Uso de Memoria

El uso de memoria se refiere a la cantidad de espacio que un algoritmo necesita para su ejecución, incluyendo variables, estructuras de datos y demás información durante la ejecución.

Técnicas de Análisis Algorítmico

Análisis Asintótico

El análisis asintótico describe cómo crece el comportamiento de un algoritmo en función del tamaño de la entrada. Las notaciones comunes son:

- **Big-O (O)**: Límite superior (peor caso).
- **Omega (Ω)**: Límite inferior (mejor caso).
- **Theta (θ)**: Límite exacto (peor y mejor caso).

Análisis del Peor Caso, Mejor Caso y Caso Promedio

- **Peor Caso**: Evalúa el rendimiento del algoritmo en la peor situación posible (máximo tiempo o espacio).
- **Mejor Caso**: Evalúa el rendimiento en la mejor situación posible (mínimo tiempo o espacio).
- **Caso Promedio**: Evalúa el rendimiento en una situación promedio, tomando en cuenta todas las entradas posibles.

Ejemplos de Análisis

Búsqueda Lineal

Código:

```
def busqueda_lineal(lista, objetivo):
```

```
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

- **Análisis:**
    - **Peor Caso:** `O(n)` - El algoritmo recorre toda la lista.
    - **Mejor Caso:** `O(1)` - El elemento está en la primera posición.
    - **Caso Promedio:** `O(n/2)` - En promedio, el algoritmo recorre la mitad de la lista.

Búsqueda Binaria
Código:

```
def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return -1
```

- **Análisis:**
    - **Peor Caso:** `O(log n)` - La lista se divide en mitades.
    - **Mejor Caso:** `O(1)` - El elemento está en el medio.
    - **Caso Promedio:** `O(log n)` - Similar al peor caso debido a la división repetitiva de la lista.

Conclusión

El análisis algorítmico es esencial para optimizar y comparar algoritmos, permitiendo crear soluciones más rápidas y eficientes. Conocer las técnicas y notaciones clave como Big-O, Omega, y Theta es fundamental para entender cómo un algoritmo se comportará a medida que las entradas crecen. En futuras clases, profundizaremos en técnicas avanzadas de análisis y optimización de algoritmos.

Tema 4: ¿Qué es la Complejidad Algorítmica?

¿Qué es la Complejidad Algorítmica?

Introducción

La **complejidad algorítmica** es un concepto clave en la ciencia de la computación que nos permite entender y comparar la eficiencia de los algoritmos. Esta se refiere a los **recursos computacionales** necesarios por un algoritmo para resolver un problema, especialmente en términos de **tiempo** y **memoria**. En esta clase, exploramos cómo se mide la complejidad algorítmica y su importancia.

Importancia de la Complejidad Algorítmica

La **complejidad algorítmica** es crucial por varias razones:

1. **Eficiencia**: Nos permite evaluar qué tan rápido es un algoritmo y seleccionar el más adecuado para un problema.
2. **Escalabilidad**: Ayuda a prever cómo se comportará un algoritmo cuando el tamaño de los datos aumenta.
3. **Comparación**: Facilita la comparación entre diferentes algoritmos que resuelven el mismo problema, permitiendo elegir la solución más eficiente.

Medición de la Complejidad Algorítmica

Notación Big-O

La **notación Big-O** describe la complejidad de un algoritmo de manera **asintótica**, es decir, cómo se comporta cuando el tamaño de la entrada **tiende al infinito**. Big-O se usa para expresar el **peor caso** de un algoritmo.

Ejemplos comunes de Big-O:

- `O(1)` **- Constante**: El tiempo de ejecución no depende del tamaño de la entrada.
- `O(log n)` **- Logarítmica**: El tiempo de ejecución crece logarítmicamente con el tamaño de la entrada.
- `O(n)` **- Lineal**: El tiempo de ejecución crece linealmente con el tamaño de la entrada.
- `O(n log n)` **- Logarítmica lineal**: El tiempo de ejecución crece linealmente con un factor logarítmico.

- `O(n^2)` - **Cuadrática**: El tiempo de ejecución crece cuadráticamente con el tamaño de la entrada.
- `O(2^n)` - **Exponencial**: El tiempo de ejecución crece exponencialmente con el tamaño de la entrada.

Ejemplos Prácticos

Ejemplo 1: Búsqueda Lineal

La **búsqueda lineal** recorre toda la lista buscando un elemento específico. En el peor caso, el algoritmo tiene que recorrer toda la lista.

Código:

python
Copiar código
```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

- **Complejidad:** En el peor caso, la complejidad es `O(n)` porque el algoritmo recorre toda la lista para encontrar el elemento o determinar que no está presente.

Ejemplo 2: Búsqueda Binaria

La **búsqueda binaria** es más eficiente que la búsqueda lineal, pero solo funciona en listas **ordenadas**. Su complejidad en el peor caso es `O(log n)`.

Código:

```
def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
```

```
    return -1
```

- **Complejidad:** La complejidad en el peor caso es `O(log n)` porque el algoritmo divide la lista a la mitad en cada iteración, reduciendo significativamente el número de comparaciones.

Comparación de Complejidades

A continuación, se presenta una tabla que muestra cómo crecen diferentes órdenes de complejidad en función del tamaño de la entrada **n**:

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(2^n)
1	1	0	1	0	1	2
10	1	3	10	30	100	1024
100	1	6	100	600	10,000	1.27E30
1,000	1	9	1,000	9,000	1,000,000	1.07E301

- En **O(1)**, el tiempo de ejecución es constante, sin importar el tamaño de la entrada.
- En **O(log n)**, el tiempo de ejecución crece lentamente a medida que `n` aumenta.
- En **O(n)**, el tiempo de ejecución crece proporcionalmente al tamaño de la entrada.
- En **O(n^2)**, el tiempo de ejecución crece cuadráticamente, lo que significa que duplicar el tamaño de la entrada cuadruplica el tiempo de ejecución.
- En **O(2^n)**, el tiempo de ejecución crece exponencialmente, lo que significa que el algoritmo se vuelve muy lento incluso para tamaños de entrada relativamente pequeños.

Conclusión

La **complejidad algorítmica** es crucial para entender la eficiencia de los algoritmos y compararlos adecuadamente. Con la notación Big-O, podemos analizar cómo un algoritmo se comporta a medida que aumenta el tamaño de la entrada y elegir la solución más eficiente para un problema dado. Al comprender la complejidad algorítmica, podemos escribir programas más **eficientes** y **escalables**.

Tema 5: ¿Qué es "Optimizar" un Algoritmo?

¿Qué es "Optimizar" un Algoritmo?

Introducción

La **optimización de algoritmos** es un proceso esencial en la ciencia de la computación, cuyo objetivo es mejorar la eficiencia de un algoritmo. Optimizar un algoritmo implica hacerlo más rápido (reduciendo su tiempo de ejecución) y/o más económico en cuanto a memoria (reduciendo su consumo de recursos). En esta clase, exploramos los conceptos clave de la optimización de algoritmos, su importancia y algunos enfoques comunes.

Definición de Optimización de Algoritmos

Optimizar un algoritmo se refiere a hacer que un algoritmo sea más eficiente en términos de **tiempo de ejecución** (velocidad) y/o **uso de memoria**. La optimización busca mejorar estos dos aspectos para resolver problemas de manera más eficaz.

Ejemplo de Optimización

Consideremos el problema de encontrar la suma de los primeros `n` números naturales.

Algoritmo No Optimizado

```
def suma_no_optimizada(n):
    suma = 0
    for i in range(1, n + 1):
        suma += i
    return suma

print(suma_no_optimizada(100))
```

- **Complejidad temporal:** O(n) – El algoritmo recorre todos los números hasta `n`.

Algoritmo Optimizado

python
Copiar código

```
def suma_optimizada(n):
    return n * (n + 1) // 2

print(suma_optimizada(100))
```

- **Complejidad temporal:** O(1) – El algoritmo realiza un solo cálculo, independientemente de n.

Aquí vemos que el algoritmo optimizado realiza la misma tarea con una complejidad de tiempo constante, lo que significa que su tiempo de ejecución no depende del tamaño de n, en comparación con el algoritmo no optimizado, que tiene una complejidad lineal.

Importancia de la Optimización

Eficiencia del Tiempo de Ejecución

La velocidad de un algoritmo es esencial, sobre todo cuando se trabaja con grandes volúmenes de datos o aplicaciones en tiempo real. Un algoritmo optimizado puede reducir significativamente el tiempo de procesamiento, lo cual es crucial en aplicaciones de alto rendimiento, como sistemas financieros o servicios web.

Uso de Memoria

Optimizar un algoritmo también implica reducir el uso de memoria. Esto es especialmente importante en sistemas con recursos limitados, como dispositivos móviles o sistemas embebidos. Un algoritmo que consume menos memoria puede mejorar el rendimiento general de un programa y evitar cuellos de botella.

Escalabilidad

Los algoritmos optimizados son más escalables, es decir, pueden manejar de manera más eficiente el aumento del tamaño de los datos. Esto es vital para aplicaciones que deben procesar grandes cantidades de información, como bases de datos, motores de búsqueda, o aplicaciones científicas.

Principios Básicos de la Optimización de Algoritmos

Análisis de Complejidad

Antes de optimizar un algoritmo, es crucial entender su **complejidad** en términos de **tiempo** y **espacio**. La **notación Big-O** se utiliza para describir cómo varía el rendimiento del algoritmo con el tamaño de la entrada.

Ejemplos de complejidad:

- **O(1):** Constante – El tiempo de ejecución no depende del tamaño de la entrada.
- **O(log n):** Logarítmica – El tiempo de ejecución crece de forma logarítmica.
- **O(n):** Lineal – El tiempo de ejecución crece linealmente.
- **O(n log n):** Logarítmica lineal – El tiempo de ejecución crece de forma lineal con un factor logarítmico.

- **O(n^2):** Cuadrática – El tiempo de ejecución crece de forma cuadrática.

Identificación de Cuellos de Botella

Un **cuello de botella** es una parte del algoritmo que consume más tiempo o recursos que el resto del proceso. Identificar estos cuellos de botella es esencial para optimizar un algoritmo. Estos puntos suelen ser los primeros candidatos a mejorar.

Algoritmos y Estructuras de Datos Apropiados

La elección adecuada de **algoritmos** y **estructuras de datos** puede impactar enormemente el rendimiento. Por ejemplo:

- Usar una **lista enlazada** en lugar de un **array** puede ser más eficiente en algunas situaciones.
- Implementar un **algoritmo de búsqueda binaria** en lugar de una **búsqueda lineal** puede reducir el tiempo de búsqueda en listas ordenadas de `O(n)` a `O(log n)`.

La correcta selección de estos elementos es fundamental para lograr una mayor eficiencia.

Técnicas de Optimización Comunes

Algunas de las técnicas de optimización más utilizadas incluyen:

- **Memoización y Caché:** Almacenar resultados de cálculos costosos para reutilizarlos cuando sea necesario, evitando cálculos repetidos.
- **Dividir y Vencer:** Dividir un problema en subproblemas más pequeños y resolverlos individualmente. Esta técnica es útil en algoritmos como la **búsqueda binaria** o el **merge sort**.
- **Eliminación de Subexpresiones Comunes:** Reutilizar los resultados de subexpresiones que se repiten en lugar de recalcularlas cada vez.

Conclusión

Optimizar un algoritmo es un proceso esencial que mejora la **eficiencia** de los programas al reducir el tiempo de ejecución y el uso de recursos. Comprender los principios básicos de la optimización, como el análisis de complejidad y la identificación de cuellos de botella, es clave para escribir código más rápido y escalable. A medida que avanzamos, exploraremos más técnicas avanzadas y estrategias de optimización que pueden ser aplicadas en diferentes contextos.

Tema 6: Iteración de Soluciones Aplicando Pensamiento Algorítmico Optimizado

Introducción

La **optimización algorítmica** es clave para mejorar la eficiencia de los programas. Implica refinar un algoritmo para que resuelva un problema de manera más rápida y con un menor uso de recursos. En este contexto, **el pensamiento algorítmico optimizado** es un enfoque iterativo en el cual, partiendo de una solución básica, se aplican mejoras de manera continua para alcanzar un rendimiento superior. A través de este proceso, es posible optimizar la complejidad temporal y de memoria, y hasta incorporar técnicas avanzadas como la paralelización.

¿Qué es el Pensamiento Algorítmico Optimizado?

El **pensamiento algorítmico optimizado** se refiere a un proceso en el cual:

1. **Identificación del problema:** Se entiende completamente el desafío a resolver.
2. **Diseño de una solución inicial:** Se crea un algoritmo básico para resolver el problema.
3. **Análisis de la eficiencia:** Se evalúa el algoritmo en términos de **tiempo de ejecución** y **uso de memoria**.
4. **Optimización:** A través de iteraciones, el algoritmo se mejora para hacerlo más eficiente.

Este proceso iterativo permite perfeccionar las soluciones, asegurando que el algoritmo sea lo más eficiente posible dentro del contexto del problema.

Importancia de la Iteración en la Optimización

La **iteración** es fundamental para mejorar un algoritmo. Al ejecutar varias iteraciones, se pueden:

- Detectar **cuellos de botella** (partes del algoritmo que consumen más recursos).
- Eliminar **redundancias** o pasos innecesarios.
- Aplicar **técnicas avanzadas** de optimización, como la paralelización o el uso de estructuras de datos más eficientes.

Esta práctica permite que el algoritmo evolucione desde una versión básica hasta una versión altamente eficiente.

Técnicas de Optimización Algorítmica

1. Mejorar la Complejidad Temporal

La **complejidad temporal** indica cuánto tiempo tarda un algoritmo en ejecutarse, dependiendo del tamaño de la entrada. Para mejorar la complejidad temporal, algunas estrategias incluyen:

- **Reemplazo de algoritmos ineficientes:** Ejemplo, usar búsqueda binaria en lugar de búsqueda lineal.
- **Uso de estructuras de datos adecuadas:** Como árboles binarios balanceados en lugar de listas enlazadas.

2. Reducir el Uso de Memoria

El **uso de memoria** se refiere a cuánta memoria necesita el algoritmo para completarse. Para reducir la memoria, se pueden aplicar técnicas como:

- **Eliminar variables innecesarias** que no aportan al resultado final.
- **Usar estructuras de datos compactas**, como arreglos estáticos frente a listas enlazadas.

3. Aprovechar la Paralelización

La **paralelización** divide el problema en subproblemas más pequeños que se pueden resolver simultáneamente. Esto puede mejorar drásticamente el rendimiento en sistemas con múltiples núcleos de procesamiento. Algunas formas de paralelización son:

- **Uso de hilos y procesos** para ejecutar tareas en paralelo.
- **Algoritmos paralelos** que resuelven grandes problemas de manera distribuida.

Ejemplos Prácticos de Optimización

Ejemplo 1: Búsqueda Lineal a Búsqueda Binaria

Código Inicial: Búsqueda Lineal

python
Copiar código

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

- **Complejidad temporal:** O(n).

Código Optimizado: Búsqueda Binaria

```
def busqueda_binaria(lista, objetivo):
    bajo = 0
    alto = len(lista) - 1
    while bajo <= alto:
        medio = (bajo + alto) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            bajo = medio + 1
        else:
            alto = medio - 1
    return -1
```

- **Complejidad temporal:** O(log n), mucho más eficiente que la búsqueda lineal para listas grandes.

Ejemplo 2: Ordenamiento por Burbuja a Quicksort

Código Inicial: Ordenamiento por Burbuja

```
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista
```

- **Complejidad temporal:** O(n^2), ineficiente para grandes listas.

Código Optimizado: Quicksort

```
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]
    return quicksort(menores) + [pivote] + quicksort(mayores)
```

- **Complejidad temporal:** O(n log n), mucho más eficiente que el ordenamiento por burbuja para listas grandes.

Proceso de Iteración y Optimización

1. **Diseño Inicial:** Crear una solución básica que resuelva el problema.
2. **Análisis Inicial:** Evaluar la eficiencia del algoritmo mediante notación Big-O.
3. **Identificación de Cuellos de Botella:** Localizar partes del código que consumen más tiempo o memoria.
4. **Aplicación de Técnicas de Optimización:** Mejorar la complejidad temporal y reducir el uso de memoria.
5. **Evaluación Post-Optimización:** Comparar el rendimiento del algoritmo optimizado con el original.
6. **Iteración Continua:** Repetir el proceso para refinar el algoritmo hasta alcanzar una eficiencia óptima.

Conclusión

El **pensamiento algorítmico optimizado** es una habilidad esencial en el desarrollo de software eficiente. A través de un proceso iterativo, es posible transformar soluciones iniciales en algoritmos altamente eficientes. La iteración y aplicación de técnicas de optimización, como la mejora de la complejidad temporal, reducción de memoria y paralelización, permite que los algoritmos sean capaces de manejar problemas cada vez más grandes y complejos.

UNIDAD 4

Tema 1: Aplicaciones Prácticas del Aprendizaje Automático

Introducción

El **aprendizaje automático** ha revolucionado una amplia gama de sectores, permitiendo mejoras significativas en la toma de decisiones y en la automatización de procesos. A través de modelos que aprenden de los datos, las aplicaciones de aprendizaje automático están transformando sectores como la medicina, la industria financiera, el comercio y muchos más.

Aplicaciones Comunes del Aprendizaje Automático

1. Procesamiento del Lenguaje Natural (NLP)

El **procesamiento del lenguaje natural** se centra en la interacción entre las máquinas y el lenguaje humano, permitiendo que las computadoras comprendan, interpreten y generen texto en lenguaje natural. Las aplicaciones incluyen:

- **Análisis de Sentimientos:** Evaluar el tono y el sentimiento en el texto (positivo, negativo o neutral).
- **Traducción Automática:** Traducir texto de un idioma a otro de forma automática.
- **Chatbots y Asistentes Virtuales:** Automáticamente generar respuestas y realizar tareas utilizando comprensión de lenguaje.

2. Visión por Computadora

La **visión por computadora** permite a las máquinas "ver" e interpretar imágenes y videos, emulando capacidades visuales humanas. Las aplicaciones incluyen:

- **Reconocimiento de Imágenes:** Identificación de objetos, personas y escenas en imágenes y videos.
- **Detección de Rostros:** Reconocer y autenticar personas a través de características faciales.
- **Diagnóstico Médico:** Analizar imágenes médicas (como rayos X o resonancias magnéticas) para identificar enfermedades.

3. Sistemas de Recomendación

Los **sistemas de recomendación** son utilizados para sugerir productos, servicios o contenidos a los usuarios en función de sus preferencias previas. Ejemplos comunes incluyen:

- **Recomendación de Productos:** Sugerencias personalizadas en plataformas de comercio electrónico.
- **Recomendación de Contenidos:** Sugerencias de películas, música o libros en servicios de streaming.
- **Recomendación de Amigos:** Sugerencias de conexiones en redes sociales, basadas en intereses comunes.

4. Detección de Fraude

El aprendizaje automático es crucial en la detección de **fraude** en diferentes sectores, ya que permite identificar patrones sospechosos a partir de grandes volúmenes de datos. Algunas aplicaciones incluyen:

- **Transacciones Financieras:** Detectar transacciones fraudulentas en tiempo real.
- **Fraude en Seguros:** Identificar reclamaciones fraudulentas basadas en patrones atípicos.
- **Ciberseguridad:** Detectar ataques cibernéticos y prevenir accesos no autorizados.

5. Predicción y Mantenimiento Predictivo

El aprendizaje automático puede prever eventos futuros y realizar **mantenimiento predictivo**, lo que optimiza la eficiencia operativa. Algunas aplicaciones incluyen:

- **Predicción de Demanda:** Estimar la demanda futura de productos y servicios.
- **Mantenimiento Predictivo:** Predecir el fallo de equipos y maquinaria antes de que ocurra.
- **Análisis de Series Temporales:** Analizar datos históricos para predecir comportamientos futuros.

Ejemplos Prácticos

Ejemplo 1: Análisis de Sentimientos

En este ejemplo, se utiliza la librería **TextBlob** para realizar un análisis de sentimientos de un texto. Este tipo de análisis puede ser utilizado en aplicaciones de atención al cliente o en la evaluación de opiniones en redes sociales.

```
from textblob import TextBlob

# Ejemplo de texto
texto = "¡Me encanta aprender sobre aprendizaje automático!"

# Análisis de sentimientos
analisis = TextBlob(texto)
sentimiento = analisis.sentiment

print("Polaridad:", sentimiento.polarity)
print("Subjetividad:", sentimiento.subjectivity)
```

- **Polaridad:** Mide si el sentimiento es positivo o negativo (valor entre -1 y 1).
- **Subjetividad:** Mide qué tan subjetivo o objetivo es el texto (valor entre 0 y 1).

Ejemplo 2: Reconocimiento de Imágenes

Aquí se utiliza un modelo preentrenado de **Keras** para realizar el reconocimiento de imágenes, que puede ser útil para aplicaciones como la clasificación de imágenes en redes sociales, diagnóstico médico, entre otros.

```
from keras.models import load_model
from keras.preprocessing import image
import numpy as np

# Cargar modelo pre-entrenado
```

```
modelo = load_model('modelo_reconocimiento_imagenes.h5')

# Cargar imagen
img = image.load_img('ejemplo.jpg', target_size=(224, 224))
img_array = image.img_to_array(img)
img_array = np.expand_dims(img_array, axis=0)

# Normalizar imagen
img_array /= 255.0

# Predicción
predicciones = modelo.predict(img_array)

# Resultado
clase_predicha = np.argmax(predicciones[0])
print("Clase predicha:", clase_predicha)
```

- **Modelo Preentrenado:** Se utiliza un modelo de red neuronal previamente entrenado para reconocer clases de imágenes.
- **Predicción:** El modelo predice la clase a la que pertenece la imagen cargada.

Conclusión

El aprendizaje automático tiene un impacto profundo en numerosas áreas, mejorando la eficiencia y capacidad de toma de decisiones a través de la automatización y el análisis inteligente de datos. Desde el **procesamiento del lenguaje natural** hasta la **visión por computadora** y la **detección de fraude**, las aplicaciones del aprendizaje automático están cambiando la forma en que interactuamos con la tecnología y facilitando la innovación en diversos campos.

Tema 2: Evaluación de Modelos de Aprendizaje Automático

Introducción

La **evaluación** de modelos de aprendizaje automático es esencial para verificar que los modelos sean precisos y generalicen correctamente a nuevos datos. Mediante el uso de métricas y técnicas específicas, podemos obtener información clave sobre el rendimiento del modelo, lo que permite ajustarlo y optimizarlo para obtener mejores resultados.

Importancia de la Evaluación

Evaluar el modelo permite:

1. Medir su rendimiento.
2. Compararlo con otros modelos.
3. Identificar áreas de mejora.
4. Garantizar que el modelo no se sobreajuste (overfitting) a los datos de entrenamiento, sino que sea capaz de generalizar a nuevos datos.

Conjunto de Datos de Evaluación

El proceso típico para evaluar un modelo consiste en:

- **Conjunto de Entrenamiento:** Utilizado para entrenar el modelo.
- **Conjunto de Prueba:** Usado para evaluar el rendimiento del modelo en datos que no ha visto antes.

Además, se puede usar **validación cruzada** para realizar una evaluación más robusta.

Evaluación de Modelos de Clasificación

1. Matriz de Confusión

La **matriz de confusión** muestra el rendimiento del modelo al clasificar ejemplos en diferentes clases, indicando cuántos fueron correctamente clasificados y cuántos no.

- **Componentes:**
    - **Verdaderos Positivos (TP):** Casos correctamente clasificados como positivos.
    - **Verdaderos Negativos (TN):** Casos correctamente clasificados como negativos.
    - **Falsos Positivos (FP):** Casos incorrectamente clasificados como positivos.
    - **Falsos Negativos (FN):** Casos incorrectamente clasificados como negativos.

2. Exactitud (Accuracy)

La **exactitud** mide la proporción de predicciones correctas sobre el total de predicciones.

Exactitud=TP+TNTP+TN+FP+FN\text{Exactitud} = \frac{TP + TN}{TP + TN + FP + FN}Exactitud=TP+TN+FP+FNTP+TN

3. Precisión (Precision)

La **precisión** mide cuántas de las predicciones positivas fueron correctas.

Precisio´n=TPTP+FP\text{Precisión} = \frac{TP}{TP + FP}Precisio´n=TP+FPTP

4. Recall (Sensibilidad)

El **recall** mide cuántos de los casos positivos fueron correctamente identificados.

Recall=TPTP+FN\text{Recall} = \frac{TP}{TP + FN}Recall=TP+FNTP

5. F1 Score

El **F1 Score** es la media armónica entre precisión y recall, proporcionando una métrica balanceada para los modelos de clasificación.

F1 Score=2×Precisio´n×RecallPrecisio´n+Recall\text{F1 Score} = 2 \times \frac{\text{Precisión} \times \text{Recall}}{\text{Precisión} + \text{Recall}}F1 Score=2×Precisio´n+RecallPrecisio´n×Recall

6. Curva ROC y AUC

La **curva ROC** muestra el trade-off entre el recall y la tasa de falsos positivos. El **AUC** (Área bajo la curva) mide el área bajo la curva ROC, indicando el rendimiento del modelo en general.

Evaluación de Modelos de Regresión

1. Error Absoluto Medio (MAE)

El **MAE** mide el error promedio absoluto entre las predicciones y los valores reales.

MAE=1n∑i=1n∣yi−yi^∣\text{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y_i}|MAE=n1i=1∑n∣yi−yi^∣

2. Error Cuadrático Medio (MSE)

El **MSE** mide el promedio de los errores al cuadrar las diferencias entre las predicciones y los valores reales.

MSE=1n∑i=1n(yi−yi^)2\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2MSE=n1i=1∑n(yi−yi^)2

3. Raíz del Error Cuadrático Medio (RMSE)

El **RMSE** es la raíz cuadrada del MSE, proporcionando una métrica que tiene las mismas unidades que los valores originales.

RMSE=1n∑i=1n(yi−yi^)2\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y_i})^2}RMSE=n1i=1∑n(yi−yi^)2

4. R^2 (Coeficiente de Determinación)

El **R^2** mide la proporción de la varianza de la variable dependiente que es explicada por el modelo.

R2=1−∑i=1n(yi−yi^)2∑i=1n(yi−y¯)2R^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y_i})^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2}R2=1−∑i=1n(yi−y¯)2∑i=1n(yi−yi^)2

Ejemplos Prácticos

Ejemplo 1: Evaluación de un Modelo de Clasificación

En este ejemplo, se utiliza un modelo de regresión logística para clasificar datos binarios y evaluar su rendimiento mediante diversas métricas.

```
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Ejemplo de datos
X = [[1], [2], [3], [4], [5]]
y = [0, 0, 1, 1, 1]

# Dividir los datos
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Crear y entrenar el modelo
modelo = LogisticRegression()
modelo.fit(X_train, y_train)

# Predicciones
y_pred = modelo.predict(X_test)

# Evaluación
matriz_confusion = confusion_matrix(y_test, y_pred)
exactitud = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred)

print("Matriz de Confusión:\n", matriz_confusion)
print("Exactitud:", exactitud)
```

```
print("Precisión:", precision)
print("Recall:", recall)
print("F1 Score:", f1)
print("ROC AUC:", roc_auc)
```

Ejemplo 2: Evaluación de un Modelo de Regresión

Aquí, se entrena un modelo de regresión lineal y se evalúa su rendimiento usando métricas como el MAE, MSE, RMSE y R^2.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression

# Ejemplo de datos
X = [[1], [2], [3], [4], [5]]
y = [1, 3, 3, 2, 5]

# Dividir los datos
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2)

# Crear y entrenar el modelo
modelo = LinearRegression()
modelo.fit(X_train, y_train)

# Predicciones
y_pred = modelo.predict(X_test)

# Evaluación
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = mean_squared_error(y_test, y_pred, squared=False)
r2 = r2_score(y_test, y_pred)

print("MAE:", mae)
print("MSE:", mse)
print("RMSE:", rmse)
print("R^2:", r2)
```

Conclusión

La **evaluación** de modelos de aprendizaje automático es un paso fundamental en el desarrollo de modelos efectivos. Usando las métricas adecuadas para clasificación y regresión, se puede evaluar de manera precisa el rendimiento del modelo, identificar sus fortalezas y debilidades, y mejorar su capacidad para generalizar a nuevos datos.

Tema 3: Fundamentos del Aprendizaje Automático

Introducción

El **aprendizaje automático** (Machine Learning) es una disciplina dentro de la inteligencia artificial que permite a las máquinas aprender de los datos sin ser explícitamente programadas para tareas específicas. Mediante el análisis de grandes cantidades de datos, los modelos de aprendizaje automático pueden identificar patrones y hacer predicciones o tomar decisiones de manera autónoma. Este campo ha transformado industrias como la tecnología, la salud, las finanzas y más.

¿Qué es el Aprendizaje Automático?

El aprendizaje automático permite que los sistemas mejoren su rendimiento en tareas a medida que se exponen a más datos. Esto se logra mediante algoritmos que analizan y extraen patrones de los datos para hacer predicciones, clasificaciones o decisiones.

En lugar de ser programados para seguir una secuencia de instrucciones predefinidas, los modelos de aprendizaje automático se entrenan utilizando datos y optimizan su capacidad para realizar tareas específicas.

Tipos de Aprendizaje Automático

1. **Aprendizaje Supervisado**
   En el aprendizaje supervisado, el algoritmo se entrena con datos etiquetados, es decir, donde cada entrada tiene una salida conocida. El objetivo es que el modelo aprenda a predecir las salidas para nuevas entradas.
   - **Ejemplo:** Clasificación de correos electrónicos en "spam" o "no spam".
   - **Algoritmos Comunes:** Regresión lineal, Máquinas de Soporte Vectorial (SVM), redes neuronales.
2. **Aprendizaje No Supervisado**
   En el aprendizaje no supervisado, los datos no tienen etiquetas. El algoritmo busca encontrar patrones o estructuras subyacentes sin la guía de una salida esperada.
   - **Ejemplo:** Agrupación de clientes en diferentes segmentos basados en sus comportamientos de compra.

    - **Algoritmos Comunes:** K-means, análisis de componentes principales (PCA), redes neuronales autoencoders.
3. **Aprendizaje por Refuerzo**
   El aprendizaje por refuerzo se centra en agentes que aprenden a través de la interacción con su entorno. El agente recibe recompensas o castigos en función de sus acciones, lo que lo impulsa a mejorar su comportamiento.
    - **Ejemplo:** Un agente jugando al ajedrez y mejorando sus estrategias a medida que juega más partidas.
    - **Algoritmos Comunes:** Q-learning, aprendizaje profundo por refuerzo.

Componentes Clave del Aprendizaje Automático

1. **Datos**
   Los datos son la base del aprendizaje automático. La calidad y cantidad de los datos afectan significativamente el rendimiento del modelo. Existen tres tipos de datos:
    - **Datos de Entrenamiento:** Utilizados para enseñar al modelo.
    - **Datos de Validación:** Usados para ajustar los parámetros del modelo.
    - **Datos de Prueba:** Utilizados para evaluar el rendimiento del modelo después del entrenamiento.
2. **Modelos**
   Un modelo es la representación matemática de un problema que se entrena con los datos. Dependiendo del tipo de problema, los modelos pueden ser:
    - **Modelos Lineales:** Como la regresión lineal.
    - **Modelos No Lineales:** Como las redes neuronales, que permiten capturar relaciones más complejas.
3. **Función de Pérdida**
   La función de pérdida mide qué tan bien está funcionando un modelo. El objetivo durante el entrenamiento es minimizar esta función. Un ejemplo común es el **Error Cuadrático Medio (MSE)**, utilizado en problemas de regresión.
4. **Algoritmos de Optimización**
   Los algoritmos de optimización ajustan los parámetros del modelo para minimizar la función de pérdida. Uno de los algoritmos más utilizados es el **descenso de gradiente**.

Ejemplo Práctico

Problema: Clasificación de Flores Iris

El conjunto de datos **Iris** es un clásico en aprendizaje automático, utilizado para clasificar flores de iris en tres especies diferentes basándose en las medidas de los sépalos y pétalos.

Pasos:

1. **Recolectar Datos:** Utilizar el conjunto de datos Iris que contiene las medidas de sépalos y pétalos, junto con las etiquetas de clase (especies de flores).

2. **Preprocesar Datos:** Limpiar y normalizar los datos para asegurarse de que todos los valores estén en la misma escala.
3. **Seleccionar un Modelo:** Elegir un modelo de clasificación, como la **regresión logística**.
4. **Entrenar el Modelo:** Dividir los datos en conjuntos de entrenamiento y prueba, luego entrenar el modelo con los datos de entrenamiento.
5. **Evaluar el Modelo:** Medir la precisión del modelo utilizando el conjunto de prueba.

Código de Ejemplo (Python):

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Cargar los datos
iris = load_iris()
X = iris.data
y = iris.target

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)

# Crear y entrenar el modelo
modelo = LogisticRegression(max_iter=200)
modelo.fit(X_train, y_train)

# Realizar predicciones
y_pred = modelo.predict(X_test)

# Evaluar el modelo
precision = accuracy_score(y_test, y_pred)
print(f'Precisión del modelo: {precision:.2f}')
```

Conclusión

El aprendizaje automático es una herramienta poderosa que está transformando diversas áreas al permitir que las máquinas tomen decisiones inteligentes a partir de datos. Los fundamentos del aprendizaje automático, como los tipos de aprendizaje, los componentes clave y los algoritmos utilizados, son esenciales para desarrollar modelos efectivos. En las siguientes clases, profundizaremos en los algoritmos específicos y las técnicas avanzadas que permiten resolver problemas más complejos.

Tema 4: Herramientas y Librerías de Aprendizaje Automático en Python

1. Scikit-Learn

Scikit-Learn es una de las librerías más utilizadas para el aprendizaje automático en Python. Ofrece una variedad de algoritmos para clasificación, regresión y clustering, además de herramientas para preprocesar datos y validar modelos.

- **Características:**
  - Algoritmos para clasificación, regresión y agrupamiento.
  - Herramientas para preprocesamiento de datos y selección de modelos.
  - Soporte para validación cruzada y evaluación de modelos.

Código de Ejemplo:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Cargar el dataset Iris
data = load_iris()
X = data.data
y = data.target

# Dividir los datos en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Crear y entrenar el modelo
modelo = RandomForestClassifier(n_estimators=100)
modelo.fit(X_train, y_train)

# Realizar predicciones
y_pred = modelo.predict(X_test)

# Evaluar el modelo
exactitud = accuracy_score(y_test, y_pred)
print("Exactitud:", exactitud)
```

---

2. TensorFlow

TensorFlow es una librería de código abierto desarrollada por Google para construir y entrenar modelos de aprendizaje automático, especialmente redes neuronales profundas.

- **Características:**
  - Soporta redes neuronales profundas (Deep Learning).
  - Compatible con CPU y GPU.
  - Herramientas para producción y despliegue de modelos.

Código de Ejemplo:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Crear un modelo secuencial
modelo = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(64, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compilar el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Resumen del modelo
modelo.summary()
```

---

3. Keras

Keras es una API de alto nivel para construir redes neuronales profundas. Está integrada con TensorFlow y es conocida por su simplicidad.

- **Características:**
  - API simple y concisa.
  - Soporte para redes neuronales convolucionales (CNN) y recurrentes (RNN).
  - Integración con TensorFlow.

Código de Ejemplo:

```
from keras.models import Sequential
from keras.layers import Dense

# Crear un modelo secuencial
modelo = Sequential()
modelo.add(Dense(64, activation='relu', input_dim=10))
modelo.add(Dense(64, activation='relu'))
modelo.add(Dense(1, activation='sigmoid'))

# Compilar el modelo
modelo.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Resumen del modelo
modelo.summary()
```

---

4. PyTorch

PyTorch es una librería de código abierto desarrollada por Facebook, conocida por su flexibilidad y facilidad para hacer prototipos rápidos.

- **Características:**
  - Ideal para investigación y experimentación.
  - Soporta redes neuronales profundas y aprendizaje reforzado.
  - Herramientas avanzadas para depuración y optimización.

Código de Ejemplo:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Definir el modelo
class ModeloSimple(nn.Module):
    def __init__(self):
        super(ModeloSimple, self).__init__()
        self.fc1 = nn.Linear(10, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 1)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
```

```
        x = torch.relu(self.fc2(x))
        x = torch.sigmoid(self.fc3(x))
        return x

# Crear una instancia del modelo
modelo = ModeloSimple()

# Definir el optimizador y la función de pérdida
optimizador = optim.Adam(modelo.parameters(), lr=0.001)
criterio = nn.BCELoss()

# Resumen del modelo
print(modelo)
```

---

5. Pandas

Pandas es una librería esencial para el manejo de datos en Python. Proporciona estructuras de datos como DataFrames para manipular datos de manera eficiente.

- **Características:**
  - Soporte para datos tabulares y heterogéneos.
  - Herramientas para limpiar y transformar datos.
  - Facilita la exploración de datos mediante funciones estadísticas.

Código de Ejemplo:

```
import pandas as pd

# Crear un DataFrame
datos = {
    'Nombre': ['Ana', 'Luis', 'Marta', 'Juan'],
    'Edad': [23, 45, 34, 25],
    'Ciudad': ['Madrid', 'Barcelona', 'Valencia', 'Sevilla']
}

df = pd.DataFrame(datos)

# Mostrar el DataFrame
print(df)

# Descripción estadística
print(df.describe())
```

---

6. NumPy

NumPy es la librería fundamental para el cálculo numérico en Python, proporcionando soporte para matrices multidimensionales y funciones matemáticas.

- **Características:**
  - Soporte para arrays y matrices multidimensionales.
  - Amplias funciones matemáticas y estadísticas.
  - Integración con otras librerías científicas.

Código de Ejemplo:

```
import numpy as np

# Crear una matriz
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Operaciones con la matriz
suma = np.sum(matriz)
media = np.mean(matriz)
desviacion_estandar = np.std(matriz)

print("Suma:", suma)
print("Media:", media)
print("Desviación Estándar:", desviacion_estandar)
```

---

7. Matplotlib y Seaborn

Matplotlib y **Seaborn** son librerías de visualización de datos. Matplotlib ofrece gráficos personalizables, mientras que Seaborn proporciona una sintaxis más sencilla para crear gráficos estadísticos.

- **Características:**
  - Matplotlib: gráficos personalizables (líneas, barras, dispersión).
  - Seaborn: gráficos estadísticos y visualizaciones más atractivas.

Código de Ejemplo:

```
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import pandas as pd

# Crear un DataFrame de ejemplo
datos = {
    'A': [1, 2, 3, 4, 5],
    'B': [5, 4, 3, 2, 1]
}
df = pd.DataFrame(datos)

# Crear un gráfico de línea con Matplotlib
plt.figure(figsize=(8, 5))
plt.plot(df['A'], df['B'], marker='o')
plt.title('Gráfico de Línea con Matplotlib')
plt.xlabel('A')
plt.ylabel('B')
plt.grid(True)
plt.show()

# Crear un gráfico de dispersión con Seaborn
plt.figure(figsize=(8, 5))
sns.scatterplot(x='A', y='B', data=df)
plt.title('Gráfico de Dispersión con Seaborn')
plt.show()
```

Conclusión

Las herramientas y librerías de Python son fundamentales para cualquier profesional en el campo del aprendizaje automático. Desde la manipulación de datos con **Pandas** y **NumPy**, hasta la construcción de modelos complejos con **TensorFlow** y **PyTorch**, estas librerías facilitan tanto el desarrollo como la implementación de soluciones inteligentes. Con ellas, los investigadores y desarrolladores pueden llevar sus ideas del concepto a la práctica de forma eficiente y efectiva.

Tema 5: Modelos Básicos de Aprendizaje Automático

Introducción

El aprendizaje automático abarca una amplia variedad de modelos que pueden ser utilizados para resolver diversos problemas. En esta clase, exploraremos algunos de los modelos más básicos y fundamentales del aprendizaje automático. Estos modelos sirven como base para comprender técnicas más avanzadas y complejas.

Modelos de Aprendizaje Supervisado

1. Regresión Lineal

La regresión lineal es uno de los modelos más simples y ampliamente utilizados en el aprendizaje automático. Se utiliza para predecir un valor continuo basado en una o más variables independientes.

- **Aplicación:** Predicción de precios de casas, ventas, etc.
- **Fórmula:** `y = β0 + β1x1 + β2x2 + ... + βnxn + ε`
- **Ejemplo de Código:**

```
from sklearn.linear_model import LinearRegression

# Ejemplo de datos
X = [[1], [2], [3], [4], [5]]
y = [1, 3, 3, 2, 5]

# Crear el modelo
modelo = LinearRegression()
modelo.fit(X, y)

# Predicción
prediccion = modelo.predict([[6]])
print(prediccion)
```

2. Regresión Logística

La regresión logística se utiliza para problemas de clasificación binaria. Predice la probabilidad de que una instancia pertenezca a una de las dos clases.

- **Aplicación:** Clasificación de correos electrónicos como spam o no spam, diagnóstico de enfermedades.
- **Fórmula:** `P(y=1|x) = 1 / (1 + e^-(β0 + β1x1 + β2x2 + ... + βnxn))`
- **Ejemplo de Código:**

```
from sklearn.linear_model import LogisticRegression

# Ejemplo de datos
X = [[1], [2], [3], [4], [5]]
y = [0, 0, 1, 1, 1]

# Crear el modelo
modelo = LogisticRegression()
modelo.fit(X, y)

# Predicción
prediccion = modelo.predict([[3.5]])
print(prediccion)
```

3. Máquinas de Soporte Vectorial (SVM)

Las SVM se utilizan tanto para clasificación como para regresión. Buscan un hiperplano que maximice el margen entre las diferentes clases.

- **Aplicación:** Clasificación de imágenes, reconocimiento de voz.
- **Fórmula:** `w • x - b = 0`
- **Ejemplo de Código:**

```
from sklearn import svm

# Ejemplo de datos
X = [[1, 2], [2, 3], [3, 3], [2, 5], [3, 6]]
y = [0, 0, 1, 1, 1]

# Crear el modelo
modelo = svm.SVC()
modelo.fit(X, y)

# Predicción
prediccion = modelo.predict([[3, 4]])
print(prediccion)
```

4. Árboles de Decisión

Los árboles de decisión se utilizan para problemas de clasificación y regresión. Dividen los datos en subconjuntos basados en la característica que proporciona la máxima información.

- **Aplicación:** Diagnóstico médico, análisis de crédito.
- **Fórmula:** Basado en la métrica de entropía o índice Gini.
- **Ejemplo de Código:**

python
Copiar código

```
from sklearn.tree import DecisionTreeClassifier

# Ejemplo de datos
X = [[0, 0], [1, 1], [0, 1], [1, 0]]
y = [0, 1, 1, 0]

# Crear el modelo
modelo = DecisionTreeClassifier()
modelo.fit(X, y)

# Predicción
prediccion = modelo.predict([[1, 1]])
print(prediccion)
```

Modelos de Aprendizaje No Supervisado

1. K-Means

K-Means es un algoritmo de agrupamiento que particiona los datos en k clusters, minimizando la variación dentro de cada cluster.

- **Aplicación:** Segmentación de clientes, compresión de imágenes.
- **Fórmula:** Minimizar la suma de distancias cuadradas entre puntos y el centroide del cluster.
- **Ejemplo de Código:**

```
from sklearn.cluster import KMeans

# Ejemplo de datos
X = [[1, 2], [1, 4], [1, 0], [4, 2], [4, 4], [4, 0]]

# Crear el modelo
modelo = KMeans(n_clusters=2)
modelo.fit(X)
```

```
# Predicción
predicciones = modelo.predict([[0, 0], [4, 4]])
print(predicciones)
```

2. Análisis de Componentes Principales (PCA)

PCA es una técnica de reducción de dimensionalidad que transforma los datos a un nuevo sistema de coordenadas donde los componentes principales tienen la máxima varianza.

- **Aplicación:** Compresión de datos, visualización de datos en alta dimensión.
- **Fórmula:** Encontrar los vectores propios de la matriz de covarianza.
- **Ejemplo de Código:**

```
from sklearn.decomposition import PCA

# Ejemplo de datos
X = [[2.5, 2.4], [0.5, 0.7], [2.2, 2.9], [1.9, 2.2], [3.1, 3.0],
[2.3, 2.7], [2, 1.6], [1, 1.1], [1.5, 1.6], [1.1, 0.9]]

# Crear el modelo
modelo = PCA(n_components=1)
modelo.fit(X)

# Transformar los datos
X_reducido = modelo.transform(X)
print(X_reducido)
```

Conclusión

Los modelos básicos de aprendizaje automático proporcionan una base sólida para abordar una variedad de problemas. Comprender estos modelos es crucial antes de avanzar hacia técnicas más complejas y especializadas. En las próximas clases, exploraremos más en profundidad cada uno de estos modelos y sus aplicaciones en escenarios del mundo real.

Tema 6: Preprocesamiento de Datos para Aprendizaje Automático

Introducción

El preprocesamiento de datos es una etapa crucial en cualquier proyecto de aprendizaje automático. Los datos en su forma cruda a menudo contienen ruido, valores faltantes y formatos inconsistentes que pueden afectar negativamente el rendimiento de los modelos. En esta clase, exploraremos las técnicas fundamentales para preparar los datos antes de entrenar un modelo de aprendizaje automático.

¿Qué es el Preprocesamiento de Datos?

El preprocesamiento de datos implica transformar los datos brutos en un formato limpio y adecuado para el análisis. Este proceso incluye la limpieza, normalización, transformación y reducción de datos, entre otras técnicas.

Pasos en el Preprocesamiento de Datos

1. Recolección de Datos

El primer paso es reunir los datos necesarios para el análisis. Los datos pueden provenir de diversas fuentes, como bases de datos, archivos CSV, APIs, etc.

2. Limpieza de Datos

La limpieza de datos implica identificar y corregir errores en los datos. Esto puede incluir:

- Eliminación de valores faltantes: Existen varias estrategias para manejar valores faltantes, como eliminarlos o imputarlos.
- Eliminación de duplicados: Asegurarse de que no haya registros duplicados en los datos.
- Corrección de errores: Identificar y corregir errores tipográficos o de formato.

3. Normalización y Escalado

La normalización y el escalado son técnicas para transformar las características numéricas en un rango común, lo cual es importante para algoritmos que son sensibles a la escala de los datos.

- Normalización: Transformar las características para que tengan una media de 0 y una desviación estándar de 1.
- Escalado: Transformar las características para que estén en un rango específico, como [0, 1].

4. Codificación de Variables Categóricas

Las variables categóricas deben ser transformadas en un formato numérico que los algoritmos de aprendizaje automático puedan entender.

- Codificación One-Hot: Crear variables binarias (0 o 1) para cada categoría.
- Codificación de Etiquetas: Asignar un número único a cada categoría.

5. División del Conjunto de Datos

Dividir el conjunto de datos en conjuntos de entrenamiento y prueba es esencial para evaluar el rendimiento del modelo.

- Conjunto de Entrenamiento: Utilizado para entrenar el modelo.
- Conjunto de Prueba: Utilizado para evaluar el rendimiento del modelo.

Ejemplo Práctico: Conjunto de Datos Iris

Paso 1: Recolección de Datos

Utilizamos el conjunto de datos Iris, que contiene información sobre el largo y ancho de los sépalos y pétalos de flores de iris.

Paso 2: Limpieza de Datos

- Eliminación de valores faltantes: Verificamos si hay valores faltantes y los imputamos si es necesario.
- Eliminación de duplicados: Verificamos si hay registros duplicados y los eliminamos.

Paso 3: Normalización y Escalado

Aplicamos normalización a las características numéricas del conjunto de datos.

Paso 4: Codificación de Variables Categóricas

La variable objetivo, que es la especie de la flor, se codifica utilizando codificación de etiquetas.

Paso 5: División del Conjunto de Datos

Dividimos el conjunto de datos en un 80% para entrenamiento y un 20% para prueba.

Conclusión

El preprocesamiento de datos es un paso esencial para garantizar que los datos sean adecuados para el análisis y que los modelos de aprendizaje automático puedan aprender de manera efectiva. A través de la limpieza, normalización y transformación de datos, podemos mejorar significativamente el rendimiento de nuestros modelos. En las próximas clases, exploraremos cómo aplicar estas técnicas a conjuntos de datos más complejos y específicos.