Argentina programa 4.0

Ministerio de Desarrollo Productivo Argentina

Secretaría de Economía del Conocimiento

PROGRAMADOR JUNIOR EN MACHINE LEARNING I

UNIVERSIDAD NACIONAL DE MISIONES

FACULTAD DE CIENCIAS EXACTAS QUÍMICAS Y NATURALES

CURSO 1: Fundamentos de Machine Learning (ML) Python - Nivel 1 -

FUNDAMENTACIÓN

Python es un lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de software, la ciencia de datos y el machine learning (ML). Los desarrolladores utilizan Python porque es eficiente y fácil de aprender, además de que se puede ejecutar en muchas plataformas diferentes.

Python es un lenguaje sencillo de leer y escribir debido a su alta similitud con el lenguaje humano. Además, se trata de un lenguaje multiplataforma de código abierto y, por lo tanto, gratuito, lo que permite desarrollar software sin límites. Python puede utilizarse en proyectos de inteligencia artificial, para crear sitios web escalables, realizar cálculos estructurales complejos con elementos finitos, y diseñar videojuegos, entre otras muchas aplicaciones

Objetivos

- Familiarizar al estudiante con el entorno y las herramientas de programación de Python.
- Desarrollar las capacidades de plasmar una propuesta de resolución a un problema sencillo planteado utilizando las herramientas planteadas.
- Identificar las ventajas del trabajo colaborativo y las herramientas disponibles.
- Promover la indagación sobre los temas planteados.

Introducción al análisis de datos

¿Qué es el análisis de datos?

El análisis de datos es el proceso de examinar, transformar y modelar datos con el fin de extraer información útil, generar conclusiones y apoyar la toma de decisiones. El análisis de datos se puede aplicar a diversos campos, como la ciencia, la ingeniería, la medicina, las finanzas, el marketing, la educación y otros. El análisis de datos implica el uso de técnicas estadísticas, matemáticas, computacionales y de visualización para explorar y comprender los datos. El análisis de datos se puede clasificar en diferentes tipos según el objetivo y el método que se emplee, como el análisis descriptivo, el análisis inferencial, el análisis predictivo y el análisis prescriptivo.

Etapas del análisis de datos: recolección, procesamiento, análisis y visualización

El análisis de datos es un proceso que consiste en transformar los datos en bruto en información útil para la toma de decisiones. Para ello, se siguen las siguientes etapas:

- Recopilación de datos: se obtienen los datos de las fuentes disponibles, como bases de datos, archivos, encuestas, etc. Se debe asegurar que los datos sean de calidad, es decir, completos, precisos, relevantes y actualizados.
- Preparación de datos: se limpian y organizan los datos, eliminando los errores, las inconsistencias, los duplicados y los valores faltantes. Se seleccionan los datos que se van a analizar y se transforman en un formato adecuado para el análisis.
- Análisis de datos: se aplican técnicas estadísticas y matemáticas para explorar, describir, modelar y predecir los datos. Se pueden utilizar herramientas como tablas dinámicas, gráficos, software especializado, etc. Se busca identificar patrones, tendencias, relaciones y anomalías en los datos.
- Acción: se interpretan los resultados del análisis y se extraen conclusiones y recomendaciones. Se comunican los hallazgos a las personas o entidades interesadas, utilizando medios como informes, presentaciones, dashboards, etc. Se toman decisiones basadas en la evidencia y se evalúan sus impactos.

Tipos de análisis de datos: descriptivo, exploratorio, inferencial y predictivo

Los datos son una fuente de información valiosa para las empresas y organizaciones que quieren mejorar sus procesos, productos y servicios. Sin embargo, para extraer el máximo provecho de los datos, es necesario aplicar diferentes tipos de análisis que permitan transformarlos en conocimiento útil y accionable.

En este artículo, vamos a describir cuatro tipos de análisis de datos que se utilizan con frecuencia en el ámbito empresarial y académico: el análisis descriptivo, el análisis exploratorio, el análisis inferencial y el análisis predictivo.

El análisis descriptivo es el tipo más básico y común de análisis de datos. Su objetivo es resumir y presentar los datos de forma sencilla y comprensible, mediante tablas, gráficos o medidas estadísticas. El análisis descriptivo no busca explicar las causas o las consecuencias de los datos, sino simplemente describir lo que ha ocurrido o lo que está ocurriendo. Por ejemplo, el análisis descriptivo puede mostrar el número de ventas, la satisfacción de los clientes o la distribución geográfica de los usuarios.

El análisis exploratorio es un tipo de análisis de datos que busca descubrir patrones, tendencias o relaciones ocultas en los datos, mediante técnicas estadísticas o gráficas. El análisis exploratorio no parte de una hipótesis previa, sino que se basa en la curiosidad y la intuición del analista para explorar los datos y generar nuevas preguntas o hipótesis. Por ejemplo, el análisis exploratorio puede revelar qué variables influyen en el comportamiento de compra, qué grupos o segmentos se pueden identificar en los datos o qué anomalías o valores atípicos se pueden detectar.

El análisis inferencial es un tipo de análisis de datos que busca probar o refutar hipótesis sobre una población a partir de una muestra de datos, mediante técnicas estadísticas que estiman el grado de confianza o significación de los resultados. El análisis inferencial no solo describe lo que ocurre en los datos disponibles, sino que también infiere lo que puede ocurrir en otros casos similares que no se han observado. Por ejemplo, el análisis inferencial puede confirmar o rechazar la efectividad de una campaña publicitaria, la diferencia entre dos grupos experimentales o la relación causal entre dos variables.

El análisis predictivo es un tipo de análisis de datos que busca predecir lo que puede ocurrir en el futuro a partir de los datos históricos o actuales, mediante técnicas estadísticas o algoritmos de aprendizaje automático que construyen modelos predictivos. El análisis predictivo no solo explica lo que ha ocurrido o lo que está ocurriendo, sino que también anticipa lo que puede ocurrir bajo ciertas condiciones o escenarios. Por ejemplo, el análisis predictivo puede estimar la demanda futura de un producto, el riesgo de abandono de un cliente o la probabilidad de éxito de una acción.

Introducción a Python para el análisis de datos

Introducción a NumPy y sus funciones para el análisis de datos numéricos

NumPy es una biblioteca de Python que ofrece herramientas para el análisis de datos numéricos. Con NumPy, se pueden crear y manipular arreglos multidimensionales, que son estructuras de datos que almacenan valores numéricos de forma eficiente. NumPy también proporciona funciones matemáticas y estadísticas para operar con los arreglos, así como métodos para leer y escribir datos desde archivos.

En este texto, se presenta una introducción a NumPy y sus principales características. Se explican los conceptos básicos de los arreglos, como su creación, atributos, indexación y división. También se muestran algunas operaciones y funciones que se pueden aplicar a los arreglos, como la transposición, el cambio de forma, la suma, el producto, la media y la desviación estándar. Finalmente, se ilustra cómo usar NumPy para trabajar con datos reales, leyendo un archivo CSV y realizando un análisis exploratorio.

Para utilizar NumPy, debemos importarla como `np`:

```
import numpy as np
```

Para crear un arreglo de NumPy, podemos usar la función `np.array()` y pasarle una lista de Python como argumento:

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

Salida:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Podemos acceder a los elementos de un arreglo usando índices, al igual que con las listas de Python:

```
print(arr[0])
print(arr[-1])
```

Salida:

```
1
5
```

También podemos modificar los elementos de un arreglo usando la asignación:

```
arr[0] = 10
print(arr)
```

Salida:

```
[10 2 3 4 5]
```

Los arreglos de NumPy tienen atributos que nos dan información sobre sus características, como el número de dimensiones (`ndim`), la forma (`shape`), el tamaño (`size`), el tipo de datos (`dtype`) y el número de bytes que ocupan en memoria (`nbytes`):

```
print(arr.ndim)
print(arr.shape)
print(arr.size)
print(arr.dtype)
print(arr.nbytes)
```

Salida:

```
1
(5,)
5
int64
40
```

Para crear una matriz de NumPy, podemos usar la misma función `np.array()` pero pasándole una lista de listas anidadas como argumento:

```
mat = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(mat)
print(type(mat))
```

Salida:

```
[[1 2 3]
[4 5 6]
[7 8 9]]
<class 'numpy.ndarray'>
```

Podemos acceder a los elementos de una matriz usando dos índices: el primero indica la fila y el segundo indica la columna:

```
print(mat[0,0])
print(mat[1,2])
print(mat[-1,-1])
```

Salida:

```
1
6
9
```

También podemos modificar los elementos de una matriz usando la asignación:

```
mat[0,0] = 10
print(mat)
```

Salida:

```
[[10 2 3]
[ 4 5 6]
[ 7 8 9]]
```

Los atributos de una matriz son los mismos que los de un arreglo, pero con valores diferentes según su forma y tamaño:

```
print(mat.ndim)
print(mat.shape)
print(mat.size)
print(mat.dtype)
print(mat.nbytes)
```

Salida:

```
2
(3, 3)
9
int64
72
```

NumPy nos ofrece muchas funciones para crear arreglos y matrices con valores específicos o aleatorios. Por ejemplo, podemos usar `np.zeros()` para crear un arreglo o una matriz con ceros, `np.ones()` para crear un arreglo o una matriz con unos, `np.full()` para crear un arreglo o una matriz con un valor constante, `np.arange()` para crear un arreglo con una secuencia de números dentro de un rango dado, `np.linspace()` para crear un arreglo con una cantidad fija de números espaciados uniformemente dentro de un intervalo dado, y `np.random.*` para crear arreglos con valores aleatorios siguiendo distintas distribuciones estadísticas.

Estas funciones son muy útiles para inicializar arreglos y matrices que luego podemos usar para realizar operaciones matemáticas, análisis de datos, visualización gráfica, aprendizaje automático y otras aplicaciones científicas. NumPy también nos permite modificar las propiedades de los arreglos y matrices creados, como su forma (`shape`), su tamaño (`size`),

su tipo de dato (`dtype`) y su orden de almacenamiento en memoria (`order`). Además, podemos acceder y modificar los elementos de los arreglos y matrices usando índices, rebanadas (`slices`), máscaras (`masks`) y vistas (`views`).

Cómo crear y manipular arreglos de NumPy

Los arreglos de numpy se pueden manipular mediante operaciones aritméticas, funciones matemáticas, indexación, rebanado (slicing), transposición, cambio de forma (reshape), concatenación, división y otras operaciones. A continuación se muestran algunos ejemplos de código:

```
c = a + b # Suma elemento a elemento de los arreglos a y b
print(c)
# [[2 4 6]
#  [5 7 9]]

d = np.sin(a) # Aplicar la función seno a cada elemento del arreglo
a
print(d)
# [0.84147098 0.90929743 0.14112001]

e = b[0, 1] # Acceder al elemento en la fila 0 y columna 1 del
arreglo b
print(e)
# 2

f = b[:, :2] # Obtener un rebanado con todas las filas y las dos
primeras columnas del arreglo b
print(f)
# [[1 2]
#  [4 5]]

g = b.T # Obtener la transpuesta del arreglo b
print(g)
# [[1 4]
#  [2 5]
#  [3 6]]

h = b.reshape(3, 2) # Cambiar la forma del arreglo b a tres filas y
dos columnas
print(h)
# [[1 2]
#  [3 4]
#  [5 6]]

i = np.concatenate([a, a]) # Concatenar el arreglo a consigo mismo
print(i)
```

```
# [1 2 3 1 2 3]

j = np.split(b, 2) # Dividir el arreglo b en dos subarreglos
print(j)
# [array([[1, 2, 3]]), array([[4, 5, 6]])]
```

Cómo realizar operaciones matemáticas y estadísticas con los arreglos

Los arreglos de numpy son objetos que almacenan datos numéricos de forma eficiente y permiten realizar operaciones matemáticas y estadísticas con ellos. Algunas de las operaciones más comunes son:

- Suma, resta, multiplicación y división de arreglos: se pueden realizar estas operaciones entre dos arreglos del mismo tamaño o entre un arreglo y un escalar. El resultado es un nuevo arreglo con el mismo tamaño que los operandos. Por ejemplo:

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = a + b # suma elemento a elemento
d = a * 2 # multiplicación por un escalar
print(c) # [5 7 9]
print(d) # [2 4 6]
```

- Funciones matemáticas: se pueden aplicar funciones matemáticas como seno, coseno, exponencial, logaritmo, etc. a los elementos de un arreglo. El resultado es un nuevo arreglo con el mismo tamaño que el original. Por ejemplo:

```
import numpy as np
a = np.array([0, np.pi/2, np.pi])
b = np.sin(a) # seno de cada elemento
c = np.exp(a) # exponencial de cada elemento
print(b) # [0. 1. 0.]
print(c) # [ 1. 23.14069263 23.14069263]
```

- Estadísticas: se pueden calcular estadísticas como la media, la mediana, la desviación estándar, el mínimo, el máximo, etc. de los elementos de un arreglo. El resultado es un valor escalar o un nuevo arreglo dependiendo de si se especifica un eje o no. Por ejemplo:

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.mean(a) # media de todos los elementos
c = np.std(a, axis=0) # desviación estándar por columnas
d = np.max(a, axis=1) # máximo por filas
```

```
print(b) # 3.5
print(c) # [1.5 1.5 1.5]
print(d) # [3 6]
```

Cómo usar funciones universales (ufunc) para aplicar operaciones vectorizadas a los arreglos

Las funciones universales (`ufunc`) son funciones que operan sobre arreglos de numpy de forma elemento por elemento, soportando la difusión de arreglos, la conversión de tipos y otras características estándar. Es decir, una `ufunc` es un envoltorio "vectorizado" para una función que toma un número fijo de entradas específicas y produce un número fijo de salidas específicas. En NumPy, las funciones universales son instancias de la clase numpy.ufunc. Muchas de las funciones integradas están implementadas en código C compilado.

Para usar una ufunc para aplicar operaciones vectorizadas a los arreglos de numpy, se puede llamar a la ufunc con los arreglos como argumentos. Por ejemplo, para sumar dos arreglos elemento por elemento, se puede usar la ufunc np.add:

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

Las ufunc también admiten argumentos de palabras clave opcionales, como `out`, `where` y `axes`, que permiten modificar el comportamiento de la ufunc. Por ejemplo, el argumento out permite especificar un arreglo de salida donde se almacenará el resultado de la ufunc:

```
c = np.zeros(3)
np.add(a, b, out=c)
array([5., 7., 9.])
c
array([5., 7., 9.])
```

Cómo generar datos aleatorios con NumPy y visualizarlos con matplotlib

Para generar datos aleatorios con NumPy, podemos usar la función np.random.rand, que devuelve un array de forma especificada con valores aleatorios entre 0 y 1, siguiendo una distribución uniforme. Por ejemplo, para generar un array de 10 elementos:

```
import numpy as np
datos = np.random.rand(10)
print(datos)
```

Salida:

```
[0.5488135  0.71518937 0.60276338 0.54488318 0.4236548  0.64589411
 0.43758721 0.891773   0.96366276 0.38344152]
```

Podemos modificar el rango de los valores aleatorios multiplicando el array por un factor o sumando una constante. Por ejemplo, para generar un array de 10 elementos entre -5 y 5:

```
datos = np.random.rand(10) * 10 - 5
print(datos)
```

Salida:

```
[-2.55627407 -1.74067808 -1.98832742 -3.27543483 -4.76190476
-1.76884571
 -4.31262186  3.9177298   4.63662761 -4.66557619]
```

También podemos generar arrays multidimensionales especificando más argumentos en la función np.random.rand. Por ejemplo, para generar una matriz de 3 filas y 4 columnas:

```
datos = np.random.rand(3, 4)
print(datos)
```

Salida:

```
[[0.4236548  0.64589411 0.43758721 0.891773  ]
 [0.96366276 0.38344152 0.79172504 0.52889492]
 [0.56804456 0.92559664 0.07103606 0.0871293 ]]
```

Para visualizar los datos aleatorios con matplotlib, podemos usar la función plt.plot, que dibuja una línea que conecta los puntos dados por las coordenadas x e y. Por ejemplo, para visualizar el array de 10 elementos entre -5 y 5 que generamos antes:

```
import matplotlib.pyplot as plt
plt.plot(datos)
plt.show()
```

Podemos personalizar el gráfico añadiendo etiquetas a los ejes, títulos, leyendas, colores, etc. Por ejemplo:

```
plt.plot(datos, color='red', label='Datos aleatorios')
plt.xlabel('Índice')
plt.ylabel('Valor')
plt.title('Ejemplo de gráfico de línea')
plt.legend()
plt.show()
```

Cómo usar la álgebra lineal y las funciones especiales de NumPy

Álgebra lineal

NumPy es un paquete de Python que ofrece funciones y objetos para la computación científica. Entre sus características se encuentran el álgebra lineal y las funciones especiales, que permiten realizar operaciones matemáticas avanzadas con matrices y vectores. En este texto, vamos a ver cómo usar estas herramientas con ejemplos de código.

El álgebra lineal es la rama de las matemáticas que estudia las propiedades de los espacios vectoriales y las transformaciones lineales entre ellos. NumPy proporciona el submódulo numpy.linalg, que contiene funciones para calcular la norma, el determinante, la inversa, el rango, los valores y vectores propios, la descomposición QR y SVD, el producto escalar, el producto matricial y el producto tensorial, entre otras operaciones. Para acceder a estas funciones, debemos importar NumPy con la convención habitual:

```
import numpy as np
```

A continuación, vamos a crear dos matrices de ejemplo usando el objeto np.array, que representa un arreglo multidimensional de datos homogéneos:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
```

Podemos calcular el producto matricial de A y B usando la función np.dot o el operador @:

```
C = np.dot(A, B)
D = A @ B
```

Ambas formas nos dan el mismo resultado:

```
C = D = array([[19, 22],
               [43, 50]])
```

Podemos calcular el determinante de una matriz usando la función np.linalg.det:

```
det_A = np.linalg.det(A)
det_B = np.linalg.det(B)
```

El resultado es:

```
det_A = np.linalg.det(A)
det_B = np.linalg.det(B)
```

Podemos calcular la inversa de una matriz usando la función np.linalg.inv:

```
inv_A = np.linalg.inv(A)
inv_B = np.linalg.inv(B)
```

El resultado es:

```
inv_A = array([[-2. ,  1. ],
               [ 1.5, -0.5]])
inv_B = array([[-4. ,  3. ],
               [ 3.5, -2.5]])
```

Podemos calcular los valores y vectores propios de una matriz usando la función np.linalg.eig:

```
val_A, vec_A = np.linalg.eig(A)
val_B, vec_B = np.linalg.eig(B)
```

El resultado es:

```
val_A = array([-0.37228132,  5.37228132])
vec_A = array([[-0.82456484, -0.41597356],
               [ 0.56576746, -0.90937671]])
val_B = array([-1.62493036, 14.62493036])
vec_B = array([[-0.76454754, -0.46301614],
               [ 0.64454885, -0.88626257]])
```

Funciones especiales

Las funciones especiales son funciones matemáticas que tienen aplicaciones en diversos campos de la ciencia y la ingeniería, como la física, la estadística o la teoría de números. NumPy proporciona el submódulo numpy.special, que contiene funciones para calcular funciones gamma, beta, error, Bessel, Legendre, Chebyshev, Hermite y Laguerre, entre otras. Para acceder a estas funciones, debemos importar NumPy con la convención habitual e importar el submódulo numpy.special:

```
import numpy as np
from numpy import special
```

A continuación, vamos a crear un vector de ejemplo usando el objeto np.array:

```
x = np.array([0.1, 0.2, 0.3])
```

Podemos calcular la función gamma de x usando la función special.gamma:

```
gamma_x = special.gamma(x)
```

El resultado es:

```
gamma_x = array([9.5135077 , 4.59084371, 2.99156828])
```

Podemos calcular la función beta de x y x+1 usando la función special.beta:

```
beta_x_x1 = special.beta(x,x+1)
```

El resultado es:

```
beta_x_x1 = array([10., 10., 10.])
```

Podemos calcular la función error de x usando la función special.erf:

```
erf_x = special.erf(x)
```

El resultado es:

```
erf_x = array([0.11246292, 0.22270259, 0.32862676])
```

Podemos calcular la función Bessel de primera especie y orden cero de x usando la función special.j0:

```
j0_x = special.j0(x)
```

El resultado es:

```
j0_x = array([0.99750156, 0.99002497, 0.97762654])
```

Podemos calcular el polinomio de Legendre de grado tres de x usando la función special.legendre:

```
leg_3_x = special.legendre(3)(x)
```

El resultado es:

```
leg_3_x = array([-1., -0.,  1.])
```

Cuestionario guia de lectura del material

1. ¿Qué es numpy y para qué se utiliza?
2. ¿Qué ventajas tiene usar numpy sobre las listas de Python para el cálculo científico?
3. ¿Qué es un array de numpy y cómo se crea?
4. ¿Qué operaciones se pueden realizar con los arrays de numpy?
5. ¿Cuáles son las funciones universales de numpy y cómo se aplican a los arrays?

Bibliografía:

1. Bagnato, J. i., (2020). Aprende Machine Learning en Español: Teoría + Práctica Python. Editorial Leanpub.
2. Britos, P. V., & García Martínez, R. (2009). Propuesta de Procesos de Explotación de Información. In XV Congreso Argentino de Ciencias de la Computación.
3. Chazallet, S. (2016). Python 3: los fundamentos del lenguaje. Ediciones ENI.
4. Geron, A., (2020). Aprende Machine Learning con Scikit-Learn, Keras y Tensor Flow: Conceptos, herramientas y técnicas para construir sistemas inteligentes. Editorial O'Reilly y Anaya
5. Hilera, J. R. y Martinez, V. J. (2000) Redes Neuronales Artificiales. Fundamentos. modelos y aplicaciones. Alfaomega Ed
6. JIMÉNEZ, R. O., (2021). Python a fondo. Editorial Marcombo
7. Kuna, H. D., Caballero, S., Rambo, A., Meinl, E., Steinhilber, A., Pautsch, J., ... & Villatoro, F. (2010). Avance en procedimientos de la explotación de información para la identificación de datos faltantes, con ruido e inconsistentes. In XII Workshop de Investigadores en Ciencias de la Computación.
8. Kuna, H., Pautsch, G., Rey, M., Cuba, C., Rambo, A., ... & Villatoro, F. (2012). Obtenido de COMPARACIÓN DE LA EFECTIVIDAD DE PROCEDIMIENTOS DE LA EXPLOTACIÓN DE INFORMACIÓN PARA LA IDENTIFICACIÓN DE OUTLIERS EN BASES DE DATOS:
9. Matthes, E. (2021) Curso intensivo de Python, 2ª edición: Introducción práctica a la programación basada en proyectos. Editorial Anaya Multimedia
10. Ochoa, M. A. (2004). Herramientas inteligentes para explotación de información. Trabajo Final Especialidad en Ingeniería de Sistemas Expertos url: https://ri.itba.edu.ar/server/api/core/bitstreams/a848d640-0277-459d-9104-b37017309d31/content