

Prácticas de Aprendizaje Automático

Parte 1: Introducción a Python



ugr

Universidad
de **Granada**



Profesor

Santiago López Tapia (Santi).

Correo: sltapia@decsai.ugr.es

Tutorías:

- Horario oficial: Lunes 10:00-11:00.
- Edificio CITIC (concertar cita por correo)
- Se puede concertar cita en otro horario por correo (avisadme con 1 día de antelación).

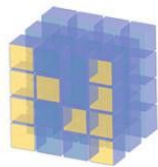


Índice

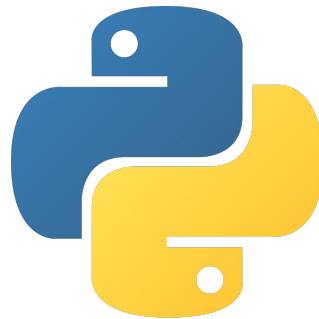
1. Python+Scikit learn. Motivos para su utilización.
2. Instalación.
3. Uso básico Anaconda.
4. Ayuda Spyder.
5. Primeros pasos.
6. Listas, tuplas y diccionarios.
7. Indexado.
8. Condicionales y bucles.
9. Funciones.
10. Clases.

Python+Scikit learn. Motivos para su utilización.

- Python:
 - Es un lenguaje de propósito general de alto nivel que permite el desarrollo rápido de aplicaciones.
 - Lenguaje interpretado que soporta programación orientada a objetos y que cuenta con una sintaxis simple e intuitiva.
 - Software libre (Python Software Foundation License).
- Scikit learn:
 - Paquete en python para análisis de datos.
 - Construido sobre NumPy, SciPy y matplotlib.
 - Software libre (licencia BSD).

























NumPy



machine learning in Python

Motivos para su utilización.

- La estructura y facilidad de uso permite implementar funciones y rutinas fácilmente según se vaya necesitando.
- Facilidad de integración en otras aplicaciones.
- Portabilidad del código (mejor que Java).
- Ejecución rápida (llamada a código compilado C).
- Scikit learn:
 - Gran cantidad de algoritmos y funciones para el tratamiento y análisis de datos, con una interfaz común y fácil de usar.
 - Muchas librerías compatibles (entre ellas para Deep learning).
 - Puede usarse en aplicaciones comerciales.
- ¡It's FREE!

Language Rank	Types	Spectrum Ranking
1. Python	 	100.0
2. C	  	99.7
3. Java	  	99.4
4. C++	  	97.2
5. C#	  	88.6
6. R		88.1
7. JavaScript	 	85.5
8. PHP		81.4
9. Go	 	76.1
10. Swift	 	75.3



Instalación



Requisitos:

- SO:
 - Windows: 7, 8, 8.1 o 10.
 - Linux: Ubuntu 16.04 LTS o superior (o derivado como Linux Mint 18 o superior).
- Python 3.6

Instalación Anaconda & IDE Spyder:

- Descargar: <https://www.anaconda.com/download/>
 - Windows -> Doble click
 - Linux -> `bash Anaconda3-5.0.1-Linux-x86_64.sh`

Instalación paquetes necesarios: `conda install scikit-learn`

Uso básico Anaconda

Crear entorno: `conda create -n <name>`

Eliminar entorno: `conda-env remove <name>`

Listar entornos: `conda-env list`

Activar/Desactivar entorno:

- Windows: `activate <name>` / `deactivate <name>`
- Linux: `source activate <name>` / `source deactivate <name>`

Instalar paquete: `conda install <name>`

Desinstalar paquete: `conda remove <name>`



Ayuda Spyder



Ctrl + i con el cursor sobre la función o método a consultar.

zeros

Definition : `zeros(shape, dtype=float, order='C')`
Type : Function of `numpy.core.multiarray` module

Return a new array of given shape and type, filled with zeros.

Parameters

shape : int or sequence of ints
Shape of the new array, e.g., (2, 3) or 2.

dtype : data-type, optional
The desired data-type for the array, e.g., `numpy.int8`. Default is `numpy.float64`.

order : {'C', 'F'}, optional
Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns

out : ndarray
Array of zeros with the given shape, dtype, and order.

See Also

`zeros_like` : Return an array of zeros with shape and type of input. `ones_like` : Return an array of ones with shape and type of input. `empty_like` : Return an empty array with shape and type of input. `ones` : Return a new array setting values to one. `empty` : Return a new uninitialized array.

Consultar código fuente: Ctrl + click izquierdo

Primeros pasos

Iniciar Spyder.

En la línea de comandos escribir código que se evalúa sobre la marcha:

```
In [1]: 2*5+10**2  
Out[1]: 110
```

Se puede realizar asignaciones, importar paquetes, etc:

```
In [2]: import numpy as np
```

```
In [3]: z = np.zeros((100, 2, 2, 2), np.float32)
```

```
In [4]: z.shape
```

```
Out[4]: (100, 2, 2, 2)
```

```
In [5]: z.sum()
```

```
Out[5]: 0.0
```

Podéis usarlo para hacer pruebas rápidas.

Primeros pasos

```
# -*- coding: utf-8 -*-
```

Necesario para introducir strings y comentarios con acentos y ñ.

```
#Declaración var y asignación
```

```
entero1 = 5 #Int
```

```
entero2 = 505 #Int
```

```
flotante = 50.5 #Float
```

```
boolean_t = True #Boolean
```

```
boolean_f = False #Boolean
```

```
string1 = 'String1' #String
```

```
string2 = 'String2' #String
```

```
#Operaciones aritméticas básicas
```

```
suma = entero1 + flotante
```

```
resta = entero1 - flotante
```

```
producto = entero1 * flotante
```

```
division = entero1 / flotante
```

```
division_entera = entero2 // entero1
```

```
resto = entero2 % entero1
```

Primeros pasos

#Operaciones lógicas

```
igual = entero2==(500 + 5)
no_igual = entero1 != suma
mayor = entero2 > entero1 # >=
menor = entero1 < entero2 # <=
and_logico = igual and mayor
or_logico = igual or no_igual
```

#Cambiar tipos

```
entero2flotante = float(entero1)
flotante2entero = int(flotante)
astring = str(entero2)
abool = bool(entero1)
```

#Strings

```
formatear = 'String con entero %d, flotante %f y string %s' % (entero1,
                                                                flotante,
                                                                string1)

concatenar = string1 + str(entero1)
```

#Mostrar por pantalla

```
print('Dos de los strings:', string1, string2)
print('String y entero:', concatenar, entero1)
```

Listas, tuplas y diccionarios.

Listas: Almacenan datos de cualquier tipo y se acceden mediante índices enteros. Son dinámicas, es decir, pueden modificarse sus elementos, añadir, eliminar, ...

Tuplas: Almacenan datos de cualquier tipo y se acceden mediante índices enteros. No pueden modificarse, solo consultarse (son estáticas).

Diccionarios: Almacenan datos de cualquier tipo y se acceden mediante palabras clave (keys). Pueden modificarse sus elementos, añadir, eliminar, ...

Listas, tuplas y diccionarios.

#Declarar tupla

```
tupla = (5, 't1', True, 0.5)
```

#Declarar lista

```
lista = [5, 't1', True, 0.5]
```

#Obtener tamaño lista y tupla

```
l_tupla = len(tupla)
```

```
l_lista = len(lista)
```

#Mostrar por pantalla

```
print(tupla)
```

```
print(lista)
```

#Acceder elemento

```
print(tupla[2])
```

```
print(lista[2])
```

```
lista[2] = 1000
```

Listas, tuplas y diccionarios.

#Añadir elemento

`lista.append(False)` *#Al final*

`lista.insert(1, 't21')` *#En la posición*

#Eliminar elemento

`lista.remove('t1')` *#Buscando*

`lista.pop()` *#Al final*

`lista.pop(1)` *#En la posición 1*

#Concatenar

`lista2 = ['a', 'b', 'c']`

`lista_combinada = lista + lista2` *#Pega la lista2 al final de lista*

#Copiar

`lista_copia = lista.copy()`

Listas, tuplas y diccionarios.

#Declarar

```
diccionario = {'a': 1, 'b': 2.0}
```

#Añadir elemento

```
diccionario['c'] = False
```

#Eliminar elemento

```
del diccionario['c']
```

#Mostrar por pantalla

```
print(diccionario)
```

#Keys

```
diccionario.keys()
```

#Values

```
diccionario.values()
```

Indexado

`lista[inicio:fin:paso]:`

- Toda la lista: `lista[:]`
- Desde inicio: `lista[inicio:]`
- Hasta fin: `lista[:fin]`
- Solo pares: `lista[::2]`

Podemos usar índices negativos, estos comenzarán a contar desde el final de la lista, Así, la posición -1 es la del último elemento de la lista.

Condicionales y bucles

#Condicional

```
if condicion:  
    #Hacer algo  
elif otra_condicion:  
    #Hacer algo  
else:  
    #Hacer algo
```

#Bucle for

```
for i in range(inicio, fin, paso):  
    #Hacer algo
```

```
for elemento in lista:  
    #Hacer algo
```

#Bucle while

```
while condicion:  
    #Hacer algo
```

Funciones

```
def funcion(a, b=1):  
    c = a+b  
  
    return c #Opcional
```

Valor por defecto para el parámetro.

```
c = funcion(1, 2) # 0 funcion(a=1, b=2)  
c_def = funcion(1)
```

Al omitirlo, b tiene el valor por defecto (1)

Tened cuidado: los parámetros pasan por referencia.

Classes

```
class Clase():  
    def __init__(self, a):  
        self.a = a  
  
    def llamar(self, b):  
        return self.a*b  
  
class Clase2(Clase):  
    def __init__(self, a, b=2.0):  
        super().__init__(a)  
        self.b = b  
  
    def llamar(self, c):  
        return self.a*self.b*c  
  
    def __call__(self, c):  
        return self.llamar(c)
```

Clases

```
c = 3  
clase2 = Clase2(a=1)  
d = clase2.llamar(c) #0 clase2(c)
```

Los parámetros también pasan por referencia. Tened cuidado con las asignaciones en los métodos que modifiquen atributos (como `__init__`).

No olvidéis el `self`.