Investigación en Inteligencia Artificial

Principios básicos del aprendizaje automático con Python

Índice

Esquema

Ideas clave

2.1. ¿Cómo estudiar este tema?

Para estudiar este tema lee las **Ideas clave** disponibles a continuación.

Sea cual sea el contexto y ámbito de nuestro proyecto, el uso de datos es una cuestión recurrente en las tareas a abarcar. Los datos no son nuestra situación de partida, el origen siempre radica en una necesidad o problema a resolver, o en una oportunidad a aprovechar. Pero todo el trabajo se realiza en base a datos de distinto tipo, y el proceso genera datos que es preciso tratar. En este tema se podrán las bases para un tratamiento adecuado de los datos en Python. Los objetivos que nos planteamos son:

- Conocer el flujo estándar de un proyecto de aprendizaje automático.
- Ser capaz de realizar un análisis descriptivo de un conjunto de datos.
- Ser capaz de procesar datos para tratar valores nulos y elementos atípicos.
- Adentrarse en el manejo de las principales librerías de manipulación de datos y cálculo numérico con Python.

Trabaja de forma autónoma con todo el código que se proporciona en las imágenes aportando y experimentando según tus propias ideas.

2.2. Principales librerías en Python para la analítica de datos

Una librería de Python es una agrupación de funcionalidades que amplían las posibilidades de Python al permitirnos aprovechar todo el potencial del lenguaje de una forma más sencilla. A continuación se muestra una lista y una breve descripción de algunas de las librerías más conocidas de Python:

NumPy: el nombre viene de Numerical Python. Es una de las librerías más útiles en Python para el cálculo matricial, implementando una gran diversidad de funciones y transformaciones algebraicas.

SciPy: el nombre corresponde a la expresión Scientific Python. SciPy evoluciona las funcionalidades de NumPy añadiendo, por ejemplo, la posibilidad de calcular transformaciones de Fourier y ejecutar procesos de optimización. Además, facilita trabajar con matrices dispersas.

Pandas: se trata de una de las librerías estrella en Python. Agrupa una serie de funcionalidades clave que facilita el siempre indispensable y engorroso trabajo de limpiar, formatear y procesar los datos adecuadamente.

Matplotlib: indispensable a la hora de generar gráficos que nos ayuden a comprender mejor las características de nuestros datos, análisis y modelos.

Scikit Learn for Machine Learning: partiendo de la base de las funcionalidades ofrecidas por NumPy, SciPy y Matplotlib, ofrece utilidades que permiten ejecutar de forma sencilla algoritmos de aprendizaje automático y modelado estadístico, incluyendo regresión, clasificación, *clustering* y reducción de dimensiones.

Statsmodels: librería enfocada en el modelado estadístico. La exploración de los datos, creación de modelos estadísticos y realización de test estadísticos es el objetivo de esta librería. Statsmodels proporciona un listado considerable de estimadores estadísticos complementados con funcionalidades para la visualización del trabajo realizado.

Seaborn: extraordinaria librería de visualización basada en Matplotlib. El objetivo de Seaborn es ayudar a las tareas de exploración de los datos, proporciona amplia y diversa información estadística de forma gráfica, lo que facilita su comprensión.

Bokeh: librería que permite la creación de gráficos interactivos y dinámicos. Bokeh es una librería eficiente que facilita la creación de cuadros de mando visualmente atractivos.

Blaze: esta librería parte de la base proporcionada tanto por NumPy como Pandas para facilitar el tratamiento de datos distribuidos y generados de forma continua (*streaming*). Permite acceder a bases de datos en distintas ubicaciones y repositorios variados como bases de datos NoSQL o ficheros de archivos distribuidos. Blaze proporciona herramientas para la visualización de datos en este contexto.

Scrapy: esta librería permite la extracción de datos a partir de páginas web.

SymPy: librería focalizada en la programación simbólica. Dada la relación con el entorno científico y técnico, SymPy incluye características que permiten trabajar cómodamente con LaTeX.

Requests y Urllib2: ambas librerías permiten interactuar con la web. urllib2 es más fácil y sencilla de manejar, no obstante, Requests puede ser una librería más recomendada para principiantes.

Además de las mencionadas, las siguientes librerías podría ser de utilidad en ciertas tipologías de proyectos:

- Os: librería que permite interactuar con el sistema operativo y sistema de archivos.
- Networkx e Igraph para la manipulación y análisis de grafos.
- Re: Regular Expressions, para trabajar con expresiones regulares en Python.
- BeutifulSoup: potente librería que permite extraer información de la web. No tiene tantas funcionalidades como Scrapy pero es bastante sencilla de manejar.

2.3. Metodología Data Science

Aunque corresponde a temas posteriores el estudio detallado del concepto de *data science* y, en especial, de la metodología de trabajo, creemos relevante hacer aquí brevemente mención a estos aspectos de cara a justificar la importancia que tiene la limpieza y preparación previa de los datos.

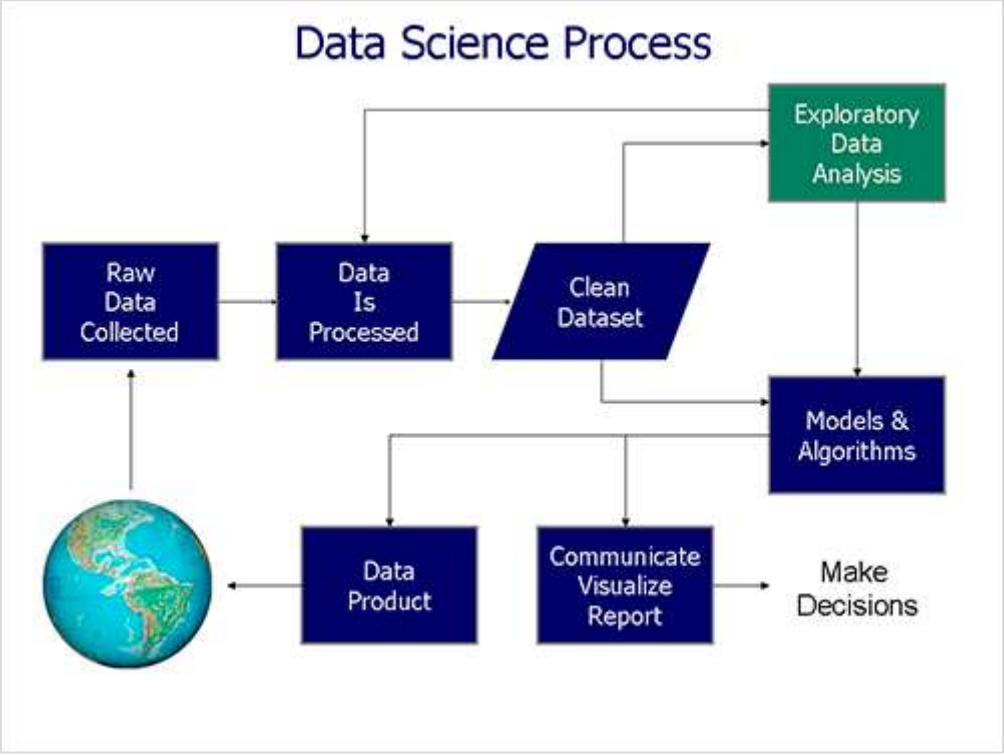


Figura 1. Data Science Process. Fuente: https://www.r-bloggers.com/a-data-science-solution-to-the-question-what-is-data-science/

La ciencia de datos o *data science* en su versión anglosajona, es una disciplina que engloba técnicas y conocimientos procedentes del área de negocio con el que se trabaje, el aprendizaje automático y las ciencias de la computación.

Su objetivo es posibilitar a las organizaciones la toma de decisiones correctas en tiempo y forma adecuados gracias a la explotación de los datos. Uno de sus focos principales se centra en intentar predecir la probabilidad de hechos futuros de forma fiable.

La ciencia de datos también se ocupa de la creación de productos innovadores basados en datos que ayuden a satisfacer las necesidades de las personas. El origen de todo el proceso es una **necesidad planteada**, un **problema** que precisa de solución o una **oportunidad** que aprovechar. El punto común es que todas estas situaciones deben abordarse gracias al uso de los datos. Es labor del experto en el área determinar qué problemas son susceptibles de resolverse bajo este enfoque.

Planteado el problema, recopilar el dato y tratarlo adecuadamente para que posibilite la generación de los modelos matemáticos correspondientes es la siguiente tarea.

El procesado, limpieza y normalización de datos es parte esencial del proceso. No solo se trata de una parte esencial sino también muy laboriosa.

En muchos proyectos, esta fase de limpieza y procesamiento de datos puede incluso implicar más de un 80% del tiempo del proyecto. Las últimas fases tienen que ver con la generación de los modelos correspondientes, explicación de resultados y puesta en producción de la solución. En los próximos temas se entrará en detalle en estos aspectos. De momento nos centraremos en la parte que nos ocupa que es el tratamiento y limpieza de datos.

Las librerías que vamos a ver a continuación son esenciales de cara a facilitarnos esta tarea.

2.4. Trabajo con NumPy

NumPy es una librería diseñada para el cálculo científico y las operaciones matemáticas con estructuras de datos matriciales. Guarda algunas semejanzas con software de parecidos propósitos como MATLAB. Al igual que se hizo en el tema anterior, ilustraremos el uso de esta librería con ejemplos varios.

Recuerda trabajar de forma autónoma evaluando todo el código que aparece en el tema y aporta tus propias ideas.

Para empezar, comprobaremos la versión de la librería que tenemos instalada.

```
import numpy as np
print ('numpy version: ', np.__version__)
```

```
numpy version:  1.13.3
```

Figura 2. Comprobando la versión de NumPy.

NumPy es una librería diseñada para trabajar con vectores multidimensionales. Los valores almacenados en la estructura de datos deben ser todos del mismo tipo. Las dimensiones de la estructura de datos reciben el nombre de "axes". Funciones útiles para probar esta librería son:

- `ndarray.ndim` – número de dimensiones de la estructura vectorial.
- `ndarray.shape` – muestra las dimensiones del vector.
- `ndarray.size` – devuelve el número total de elementos.
- `ndarray.dtype` – devuelve el tipo de los valores contenidos en el vector.
- `ndarray.itemsize` – devuelve el tamaño en bytes de los elementos del vector teniendo en cuenta su tipo.

```
a = np.array([[0,1,2,3], [4,5,6,7], [8,9,10,11]])
rows, cols = np.shape(a)
print ('Rows:{0:03d} ; Cols:{0:03d}'.format(rows, cols))
print(a)
```

```
Rows:003 ; Cols:003
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Figura 3. Dimensiones de un vector.

Podemos contemplar una gran variedad de tipos en los elementos del vector, incluso valores complejos.

```
b = np.array([[2,3], [6,7]], dtype=np.complex64)
print (b)
```

```
[[ 2.+0.j  3.+0.j]
 [ 6.+0.j  7.+0.j]]
```

Figura 4. Vectores de números complejos.

En ocasiones es de especial utilidad inicializar vectores de unas dimensiones concretas a un valor que puede ser una constante como 0 o 1, o números aleatorios.

```
np.zeros((3,4))
```

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

Figura 5. Inicialización de un vector a ceros.

La figura siguiente muestra ejemplos de otras funciones, se copia tipos y dimensiones en el primer caso, valores vacíos en el segundo caso (y se muestran los valores que existan en la memoria en ese momento) y, por último, una matriz con unos en la diagonal principal y ceros en el resto.

```
np.zeros_like(b)
```

```
array([[ 0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j]], dtype=complex64)
```

```
np.empty((2, 3))
```

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

```
np.eye(3)
```

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

Figura 6. Otras inicializaciones útiles.

Prueba ahora a ejecutar en tu equipo el siguiente código e intenta determinar la función de cada uno:

```
np.diag(np.arange(5))
```

```
array([[0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4]])
```

```
np.arange(5)
```

```
array([0, 1, 2, 3, 4])
```

```
np.tile(np.array([[6, 7], [8, 9]]), (2, 2))
```

```
array([[6, 7, 6, 7],
       [8, 9, 8, 9],
       [6, 7, 6, 7],
       [8, 9, 8, 9]])
```

Figura 7. Creando vectores multidimensionales con NumPy.

`np.arange()` es una función habitual para crear secuencias numéricas. `reshape()` es útil para ajustar las dimensiones de la estructura. La siguiente imagen muestra un ejemplo:

```
x = np.arange(4).reshape(2,2)
print(x)
```

```
[[0 1]
 [2 3]]
```

Figura 8. Ejemplo con `np.arange()` y `np.reshape()`.

No es necesario especificar los valores del vector uno a uno, también es posible recurrir a generadores como el siguiente.

```
# Use List comprehention to create a matrix
c = np.array([[10*j+i for i in range(3)] for j in range(4)])
print (c)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Figura 9. Inicializando los valores según unas iteraciones.

Para añadir una nueva dimensión o convertir un vector fila en columna, utilizamos las siguientes funciones:

```
#Genera 5 puntos equiespaciados entre 0 y 12
d = np.linspace(0, 12, 5)
print (d)
print (d[:, np.newaxis])      # make into a column vector
```

```
[  0.   3.   6.   9.  12.]
[[  0.]
 [  3.]
 [  6.]
 [  9.]
 [ 12.]]
```

Figura 10. añadiendo una nueva dimensión.

MATLAB ha sido uno de los lenguajes de referencia para el cálculo matricial. La función de `grid` de MATLAB activa líneas de cuadrícula genérico en 2-D cuando se pretende dibujar una zona del espacio. Con la función `meshgrid` de MATLAB el usuario determina completamente las líneas de cuadrícula horizontales y verticales que

aparecen en una gráfica. En Python, esta función encuentra su equivalente con el ejemplo de debajo:

```
X, Y = np.mgrid[0:5, 0:5] # similar to meshgrid in MATLAB
print(X)
print("---")
print(Y)
```

```
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
---
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
```

Figura 11. Meshgrid en Python.

En diversas situaciones, se generan matrices de dimensión considerable donde la mayor parte de los valores son cero. Almacenar dicha información según los procedimientos habituales sería ineficiente debido a que almacenaría un gran número de nulos (la inmensa mayoría). Por ello surgen estructuras de datos particulares para este caso.

```
from scipy import sparse
X = np.random.random((5, 6)) # Create an array with many zeros
X[X < 0.85] = 0
print (X)
X_csr = sparse.csr_matrix(X) # turn X into a csr (Compressed-Sparse-Row) matrix
print (X_csr)
```

```
[[ 0.95983348  0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]]
  (0, 0)        0.959833475452
```

```
print (X_csr.toarray())       # convert back to a dense array
```

```
[[ 0.95983348  0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]]
```

Figura 12. Trabajando con matrices dispersas.

NumPy permite la generación de números aleatorios en base a una semilla previamente especificada. Aunque no es obligatorio, fijar de forma previa la semilla es útil para conseguir la replicabilidad de los resultados.

```
np.random.seed(12345) # for reproducible results
np.random.rand(4,5) # uniform random numbers in [0,1]
```

```
array([[ 0.92961609,  0.31637555,  0.18391881,  0.20456028,  0.56772503],
       [ 0.5955447 ,  0.96451452,  0.6531771 ,  0.74890664,  0.65356987],
       [ 0.74771481,  0.96130674,  0.0083883 ,  0.10644438,  0.29870371],
       [ 0.65641118,  0.80981255,  0.87217591,  0.9646476 ,  0.72368535]])
```

```
np.random.randn(4,5) # standard normal distributed random numbers
```

```
array([[-0.13744673, -0.50501177, -0.19490212, -0.64476486, -0.66621581],
       [-0.8010091 , -0.55033146, -1.44937863,  0.51439326,  0.77498173],
       [ 0.86244852,  1.31726513, -0.06673705, -0.18677632, -0.14229297],
       [-0.26099421,  0.02374644, -0.64653913, -0.44688591, -0.97346678]])
```

Figura 13. Inicialización de matrices con números aleatorios.

En otras ocasiones, es útil realizar ciertas transformaciones de tipos, pasar de decimal en coma flotante a entero, de entero a carácter, etc. También podemos necesitar redondear un número.

```
a = np.array([1.7, 1.2, 1.6])
b = a.astype(int)           # <-- truncates to integer
b
```

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

Rounding:

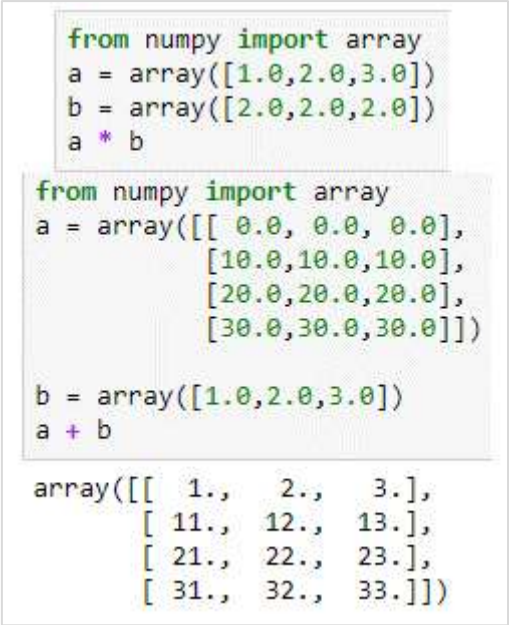
```
a = np.array([1.2, 1.5, 1.6, 2.5, 3.5, 4.5])
b = np.around(a)
print (b)                    # still floating-point
c = np.around(a).astype(int)
print (c)
```

```
[ 1.  2.  2.  2.  4.  4.]
[1 2 2 2 4 4]
```

Figura 14. Cambio de tipos y redondeo.

Es interesante que trabajes con la conversión de tipos de carácter a numérico y viceversa.

El término *«broadcasting»* describe la forma en la que NumPy trata a los vectores de diferentes dimensiones durante la ejecución de ciertas operaciones. En términos generales, el vector de menor dimensión es distribuido sobre el largo para conseguir dimensiones compatibles. Esto evita la creación de copias de los datos y favorece la creación de programas más eficientes. No obstante, es una técnica que requiere emplearse con cuidado ya que a veces puede ser perjudicial su uso, como ante ciertas operaciones binarias de funcionamiento particular previamente definidas por el usuario.



```
from numpy import array
a = array([1.0,2.0,3.0])
b = array([2.0,2.0,2.0])
a * b
```

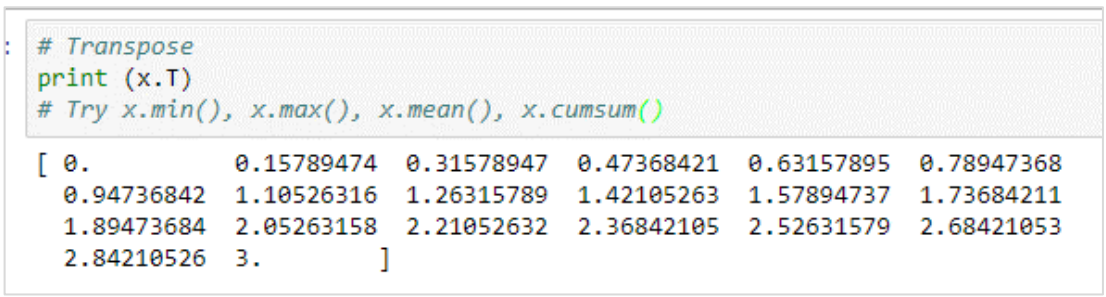
```
from numpy import array
a = array([[ 0.0, 0.0, 0.0],
           [10.0,10.0,10.0],
           [20.0,20.0,20.0],
           [30.0,30.0,30.0]])

b = array([1.0,2.0,3.0])
a + b
```

```
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```

Figura 15. *Broadcasting* in Python.

Las operaciones del álgebra lineal básica se realizan de forma muy sencilla con NumPy, pongamos algunos ejemplos.



```
# Transpose
print (x.T)
# Try x.min(), x.max(), x.mean(), x.cumsum()
```

```
[ 0.         0.15789474  0.31578947  0.47368421  0.63157895  0.78947368
  0.94736842  1.10526316  1.26315789  1.42105263  1.57894737  1.73684211
  1.89473684  2.05263158  2.21052632  2.36842105  2.52631579  2.68421053
  2.84210526  3.        ]
```

Figura 16. Matrices transpuestas en Python.

```
print (x*5)         # Scalar expansion
print (x+3)
```

```
[  0.          0.78947368   1.57894737   2.36842105   3.15789474
   3.94736842   4.73684211   5.52631579   6.31578947   7.10526316
   7.89473684   8.68421053   9.47368421  10.26315789  11.05263158
  11.84210526  12.63157895  13.42105263  14.21052632  15.        ]
[ 3.          3.15789474  3.31578947  3.47368421  3.63157895  3.78947368
  3.94736842  4.10526316  4.26315789  4.42105263  4.57894737  4.73684211
  4.89473684  5.05263158  5.21052632  5.36842105  5.52631579  5.68421053
  5.84210526  6.        ]
```

```
print (x*x.T)       # Elementwise product
print (np.dot(x,x.T))  # Dot (matrix) product
```

```
[ 0.          0.02493075  0.09972299  0.22437673  0.39889197  0.6232687
  0.89750693  1.22160665  1.59556787  2.01939058  2.49307479  3.0166205
  3.5900277   4.2132964   4.88642659  5.60941828  6.38227147  7.20498615
  8.07756233  9.        ]
61.5789473684
```

Figura 17. Suma y productos varios con NumPy.

Para el cálculo del determinante y de la matriz inversa hay que acudir a la librería SciPy.

```
from scipy import linalg
arr = np.array([[1, 2],
                [3, 4]])
linalg.det(arr)
```

```
-2.0
```

Figura 18. Cálculo del determinante.

```
print (linalg.inv(arr))
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

Figura 19. Cálculo de la matriz inversa.

El acceso a los datos es bastante intuitivo en función de la indexación de los datos, veamos algunos ejemplos.

```
# Indexing single elements
print(b[0, 0])
print(b[-1, -1]) # circular
print(b[:, 1])
```

```
0
7
[1 5]
```

Figura 20. Accediendo a los datos.

```
a = np.array([[10*j+i for i in range(6)] for j in range(6)])

#Indexing multiple elements
print(a)
print("---")
print(a[0,3:5])     # Orange
print("---")
print(a[4:,4:])     # Blue
print("---")
print(a[:, 2])      # Red
print("---")
print(a[2::2, ::2]) # Green
```

```
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
---
[3 4]
---
[[44 45]
 [54 55]]
---
[ 2 12 22 32 42 52]
---
[[20 22 24]
 [40 42 44]]
```

Figura 21. Otras formas de acceso a datos.

A la hora de copiar una matriz con todos sus atributos, se puede ejecutar la siguiente instrucción:

```
c = np.array(a, copy=True)
```

Figura 22. Copia de una matriz.

Como ejercicio, puedes comprobar las diferencias existentes entre la sentencia anterior y una asignación simple del tipo c = a

Las siguientes referencias se han empleado para elaborar la presente documentación y pueden ser útiles para avanzar en el conocimiento de la librería:

https://docs.scipy.org/doc/numpy/reference/

https://github.com/addfor/tutorials

2.5. Trabajo con Pandas

Es turno ahora de explicar los contenidos fundamentales de Panda. Nos basaremos en la introducción oficial a la librería, que puedes encontrar en: http://pandas.pydata.org/pandas-docs/stable/10min.html

Una serie de Panda es un vector unidimensional sujeto a un índice que puede especificarse o no. Para crear una serie básica empleamos la siguiente función.

```
s = pd.Series([1,3,5,np.nan,6,8])

s

0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Figura 23. Series en Panda.

No obstante, quizá la estructura de datos más usada de Panda sean los **DataFrames**. Los DataFrames se corresponden con estructuras bidimensionales donde las columnas están etiquetadas con su valor y pueden ser de tipos distintos. Puede pensarse en un DataFrame como algo similar a una hoja de cálculo de Excel. De forma opcional, un DataFrame puede tener un índice, que corresponderá a los nombres que

se desea para las filas. A continuación se muestra un ejemplo de un DataFrame donde el índice corresponde a fechas.

```
dates = pd.date_range('20130101', periods=6)
```

```
dates
```

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
```

```
df
```

	A	B	C	D
2013-01-01	-0.452164	-0.282374	1.389800	0.566133
2013-01-02	-0.401422	-0.164611	0.389598	1.711821
2013-01-03	1.767650	-0.524596	1.813586	-0.378671
2013-01-04	-0.731068	0.411821	-0.665710	-0.708695
2013-01-05	0.137137	1.497636	-1.154281	-0.068678
2013-01-06	-0.014340	-2.704823	0.457099	0.387548

Figura 24. Ejemplo de DataFrame en Pandas.

El siguiente es un ejemplo de un DataFrame con tipos de datos variados.

```
df2 = pd.DataFrame({'A':1.,
                    'B':pd.Timestamp('20130102'),
                    'C':pd.Series(1,index=list(range(4)),dtype='float32'),
                    'D':np.array([3]*4,dtype='int32'),
                    'E':pd.Categorical(["test","train","test","train"]),
                    'F':'foo'})
```

```
df2
```

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

Figura 25. Un DataFrame con tipos de datos variados.

A continuación se muestran diversas funciones útiles para acceder a las características de un DataFrame:

```
df.index
```

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
df.columns
```

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

```
df.values
```

```
array([[-0.45216355, -0.28237379,  1.38980042,  0.56613288],
       [-0.40142158, -0.1646106 ,  0.3895977 ,  1.71182139],
       [ 1.76765037, -0.52459642,  1.81358562, -0.37867051],
       [-0.73106769,  0.41182122, -0.66570989, -0.70869523],
       [ 0.1371372 ,  1.49763583, -1.15428079, -0.06867753],
       [-0.01433992, -2.70482283,  0.45709925,  0.38754771]])
```

Describe shows a quick statistic summary of your data

```
df.describe()
```

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.050966	-0.294491	0.371682	0.251576
std	0.897560	1.384937	1.142630	0.856930
min	-0.731068	-2.704823	-1.154281	-0.708695
25%	-0.439478	-0.464041	-0.401883	-0.301172

Figura 26. Utilidades para describir el dataset.

Realizar la transposición del DataFrame y ordenar son algunas de las funcionalidades más recurridas:

```
df.T
```

	2013-01-01 00:00:00	2013-01-02 00:00:00	2013-01-03 00:00:00	2013-01-04 00:00:00	2013-01-05 00:00:00	2013-01-06 00:00:00
A	-0.452164	-0.401422	1.767650	-0.731068	0.137137	-0.014340
B	-0.282374	-0.164611	-0.524596	0.411821	1.497636	-2.704823
C	1.389800	0.389598	1.813586	-0.665710	-1.154281	0.457099
D	0.566133	1.711821	-0.378671	-0.708695	-0.068678	0.387548

Sorting by an axis

```
df.sort_index(axis=1, ascending=False)
```

	D	C	B	A
2013-01-01	0.566133	1.389800	-0.282374	-0.452164
2013-01-02	1.711821	0.389598	-0.164611	-0.401422
2013-01-03	-0.378671	1.813586	-0.524596	1.767650
2013-01-04	-0.708695	-0.665710	0.411821	-0.731068
2013-01-05	-0.068678	-1.154281	1.497636	0.137137
2013-01-06	0.387548	0.457099	-2.704823	-0.014340

Figura 27. Transposición y ordenación.

El acceso a los datos guarda algunas similitudes con lo visto con la librería NumPy. Las funciones `head()` y `tail()` nos proporcionan una visión de los datos de cabeza y del final de la estructura.

```
df.head()
```

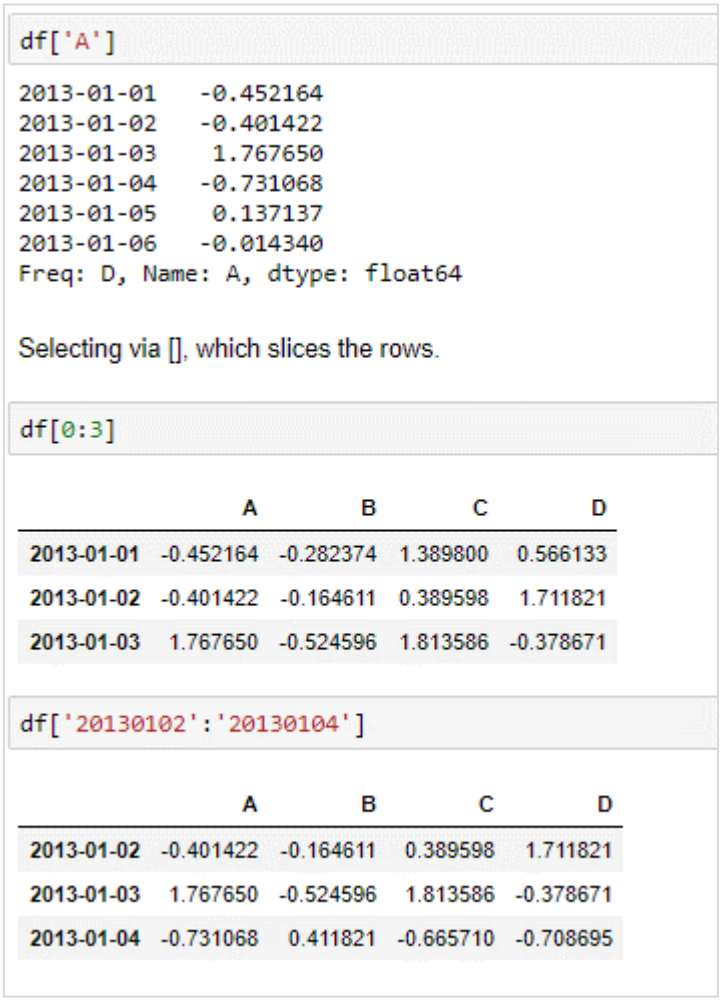
	A	B	C	D
2013-01-01	-0.452164	-0.282374	1.389800	0.566133
2013-01-02	-0.401422	-0.164611	0.389598	1.711821
2013-01-03	1.767650	-0.524596	1.813586	-0.378671
2013-01-04	-0.731068	0.411821	-0.665710	-0.708695
2013-01-05	0.137137	1.497636	-1.154281	-0.068678

```
df.tail(3)
```

	A	B	C	D
2013-01-04	-0.731068	0.411821	-0.665710	-0.708695
2013-01-05	0.137137	1.497636	-1.154281	-0.068678
2013-01-06	-0.014340	-2.704823	0.457099	0.387548

Figura 28. Encabezado y final de un DataFrame.

Para acceder a los valores individuales por columna, fila, o un rango de columnas o filas hay varias opciones disponibles que se muestran en los ejemplos siguientes.



```
df['A']
```

```
2013-01-01   -0.452164
2013-01-02   -0.401422
2013-01-03    1.767650
2013-01-04   -0.731068
2013-01-05    0.137137
2013-01-06   -0.014340
Freq: D, Name: A, dtype: float64
```

Selecting via [], which slices the rows.

```
df[0:3]
```

	A	B	C	D
2013-01-01	-0.452164	-0.282374	1.389800	0.566133
2013-01-02	-0.401422	-0.164611	0.389598	1.711821
2013-01-03	1.767650	-0.524596	1.813586	-0.378671

```
df['20130102':'20130104']
```

	A	B	C	D
2013-01-02	-0.401422	-0.164611	0.389598	1.711821
2013-01-03	1.767650	-0.524596	1.813586	-0.378671
2013-01-04	-0.731068	0.411821	-0.665710	-0.708695

Figura 29. Accediendo a los valores del DataFrame.

Si queremos realizar una selección a través del índice deberemos emplear la función `.loc()`

```
df.loc[dates[0]]
```

```
A   -0.452164
B   -0.282374
C    1.389800
D    0.566133
Name: 2013-01-01 00:00:00, dtype: float64
```

Selection by Label

```
df.loc[:,['A','B']]
```

	A	B
2013-01-01	-0.452164	-0.282374
2013-01-02	-0.401422	-0.164611
2013-01-03	1.767650	-0.524596
2013-01-04	-0.731068	0.411821
2013-01-05	0.137137	1.497636
2013-01-06	-0.014340	-2.704823

Showing label slicing, both endpoints are included

```
df.loc['20130102':'20130104',['A','B']]
```

	A	B
2013-01-02	-0.401422	-0.164611
2013-01-03	1.767650	-0.524596

Figura 30. Seleccionando valores a través del índice.

`iloc()` es otra de las opciones posibles, observar cómo en este caso es preciso pasar un número entero asociado a la posición del índice.

```
df.iloc[3]
```

```
A   -0.731068
B    0.411821
C   -0.665710
D   -0.708695
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python

```
df.iloc[3:5,0:2]
```

	A	B
2013-01-04	-0.731068	0.411821
2013-01-05	0.137137	1.497636

By lists of integer position locations, similar to the numpy/python style

```
df.iloc[[1,2,4],[0,2]]
```

	A	C
2013-01-02	-0.401422	0.389598
2013-01-03	1.767650	1.813586
2013-01-05	0.137137	-1.154281

Figura 31. Accediendo a través de la posición del índice.

También tenemos la opción de filtrar y seleccionar únicamente los valores que cumplan una determinada condición.

```
df[df.A > 0]
```

	A	B	C	D
2013-01-03	1.767650	-0.524596	1.813586	-0.378671
2013-01-05	0.137137	1.497636	-1.154281	-0.068678

A where operation for getting.

```
df[df > 0]
```

	A	B	C	D
2013-01-01	NaN	NaN	1.389800	0.566133
2013-01-02	NaN	NaN	0.389598	1.711821
2013-01-03	1.767650	NaN	1.813586	NaN
2013-01-04	NaN	0.411821	NaN	NaN
2013-01-05	0.137137	1.497636	NaN	NaN
2013-01-06	NaN	NaN	0.457099	0.387548

Figura 32. Filtros con panda.

Los usuarios acostumbrados al manejo de SQL echarán en falta funciones que permitan cruzar dos DataFrames. Afortunadamente, Pandas nos proporciona la función `join()` para este propósito.

```
left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
```

```
right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
```

```
left
```

	key	lval
0	foo	1
1	foo	2

```
right
```

	key	rval
0	foo	4
1	foo	5

```
pd.merge(left, right, on='key')
```

	key	lval	rval
0	foo	1	4
1	foo	1	5
2	foo	2	4

Figura 33. `join` con Pandas.

Y también tenemos la posibilidad de ejecutar funciones de agrupación al estilo de SQL como se muestra en este ejemplo.

```
df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'foo'],
                   'B' : ['one', 'one', 'two', 'three','two', 'two', 'one', 'three'],
                   'C' : np.random.randn(8),
                   'D' : np.random.randn(8)})
```

```
df
```

	A	B	C	D
0	foo	one	-0.754204	-1.223493
1	bar	one	1.053591	-1.032393
2	foo	two	-0.606310	-1.164652
3	bar	three	-1.004131	-0.364089
4	foo	two	-0.052413	1.515974
5	bar	two	-1.119658	-0.989549
6	foo	one	-1.705347	0.073964
7	foo	three	-1.920826	-0.542647

Grouping and then applying a function sum to the resulting groups.

```
df.groupby('A').sum()
```

A	C	D
bar	-1.070197	-2.386031
foo	-5.039101	-1.340853

Figura 34. group by en Pandas.

Como podemos apreciar, Pandas es una librería versátil, práctica y útil. Su dominio es indispensable. Por eso insistimos en la importancia de probar y validar todo el código aquí mostrado intentando insertar variaciones y nuevas pruebas.

2.6. Regresión lineal con Python

Para finalizar el tema, proponemos un ejemplo sencillo de un modelo de regresión lineal con Python. Nos apoyamos en este caso en la funcionalidad de la librería Sklearn. El modelo que vamos a ver emplea una aproximación por el método de los mínimos cuadrados. El objetivo es intentar obtener la ecuación lineal que minimiza el error cuadrático. Esta ecuación constará de una variable dependiente y un conjunto de variables independientes. Para cada x se obtendrá un error concreto considerando la distancia existente entre el punto marcado por la recta y el valor real.

Tomar potencias de dos nos ayuda a considerar el conjunto de todos los errores sin tener en cuenta si el valor real está por encima o debajo de la recta.

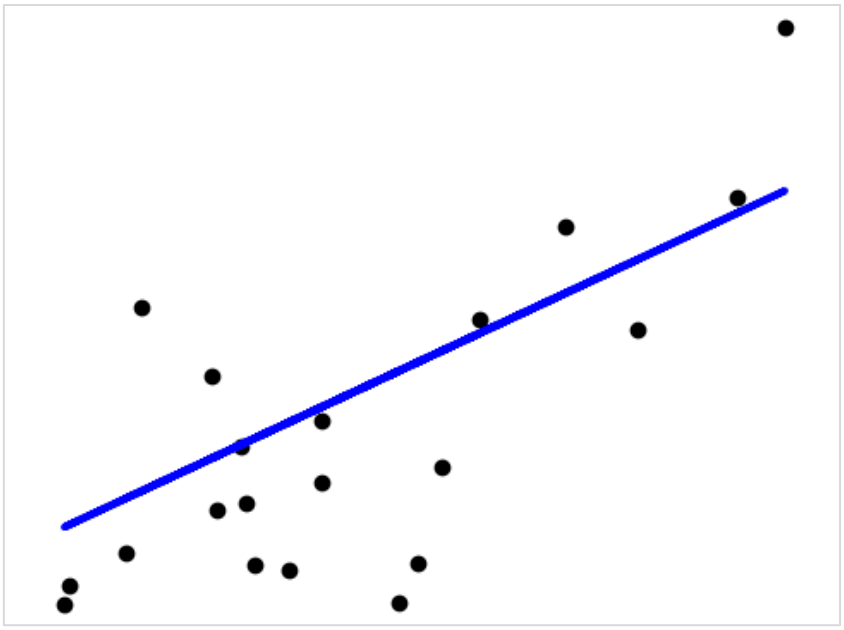


Figura 35. Ejemplo básico de regresión. Fuente: http://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html

El código está basado en unos datos sobre pacientes diabéticos de libre difusión, y corresponde al manual básico de la librería Sklearn.

Accede al manual a través del aula virtual o desde la siguiente dirección web:
http://scikit-learn.org/stable/auto_examples/linear_model/plot_ols.html

```
print(__doc__)


# Code source: Jaques Grobler
# License: BSD 3 clause


import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score

# Load the diabetes dataset
diabetes = datasets.load_diabetes()


# Use only one feature
diabetes_X = diabetes.data[:, np.newaxis, 2]

# Split the data into training/testing sets
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]

# Split the targets into training/testing sets
diabetes_y_train = diabetes.target[:-20]
diabetes_y_test = diabetes.target[-20:]
```

Figura 36. Preparando los datos para ejecutar el modelo.

En el código anterior se realiza un paso muy importante además de cargar el modelo. Si nos fijamos, se separa el conjunto de datos previos en un conjunto de entrenamiento y un conjunto de test, el primero suele ser más amplio que el segundo.

Por ejemplo, el conjunto de *training* podría contener el 70 u 80 % de la muestra, dejando el resto al conjunto de test. Sin embargo, estos porcentajes no son ciencia cierta, pues existen diversas casuísticas en las que pueden ser recomendables otras formas de distribuir los datos. Se trata esta de una estrategia muy importante en aprendizaje automático que será explicada con detalle en diferentes asignaturas y puntos del programa.

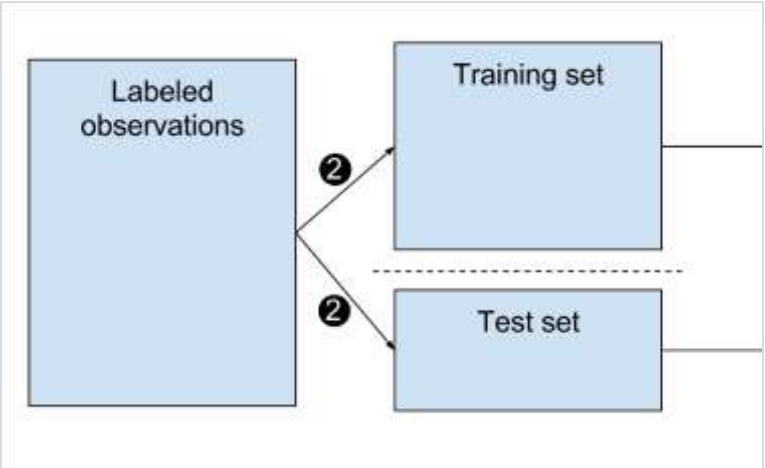


Figura 37: división de datos previos. Fuente:

https://upload.wikimedia.org/wikipedia/commons/thumb/8/88/Machine_learning_nutshell_--_Split_into_train-test_set.svg/2000px-Machine_learning_nutshell_--_Split_into_train-test_set.svg.png?

Se emplea el conjunto de entrenamiento para obtener el modelo. Este modelo se prueba sobre el conjunto de test. De esta forma sometemos al modelo a un punto de revisión adicional al considerar su rendimiento sobre unos datos que no son los empleados para aprender sus parámetros.

> Podría ponerse una analogía con un examen de matemática en el colegio. Los alumnos aprenden la lección en base a una serie de ejercicios prácticos. Si en el examen, el profesor plantea exactamente los mismos ejercicios, no podrá distinguir entre los alumnos que han comprendido la lección y los que han memorizado las respuestas a los ejercicios. Por ello, el profesor propone unos ejercicios «parecidos» a los tratados en clase pero, al mismo tiempo, distintos.

```
# Create linear regression object
regr = linear_model.LinearRegression()

# Train the model using the training sets
regr.fit(diabetes_X_train, diabetes_y_train)

# Make predictions using the testing set
diabetes_y_pred = regr.predict(diabetes_X_test)

# The coefficients
print('Coefficients: \n', regr.coef_)
# The mean squared error
print("Mean squared error: %.2f"
      % mean_squared_error(diabetes_y_test, diabetes_y_pred))
# Explained variance score: 1 is perfect prediction
print('Variance score: %.2f' % r2_score(diabetes_y_test, diabetes_y_pred))

# Plot outputs
plt.scatter(diabetes_X_test, diabetes_y_test,  color='black')
plt.plot(diabetes_X_test, diabetes_y_pred, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```

Figura 38. Obtención del modelo de regresión lineal y validación.

De momento y a modo de ejercicio, planteamos al alumno su ejecución y trabajo autónomo para intentar comprender el código introducido.

2.7. Referencias bibliográficas

McKinney, W. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. Estados Unidos: O'Reilly Media.

+ Información

A fondo

A Complete Tutorial to Learn Data Science with Python from Scratch

Jain, K. (14 de enero de 2016). A Complete Tutorial to Learn Data Science with Python from Scratch. *Analytics Vidhya.*

En el documento se muestra una introducción al sistema de citas bibliográficas incluyendo ejemplos variados.

Accede al texto a través del aula virtual o desde la siguiente dirección web:
https://www.analyticsvidhya.com/blog/2016/01/complete-tutorial-learn-data-science-python-scratch-2/

Actividades

Trabajo: Preparación de datos con Python

El objetivo de esta actividad es continuar familiarizándonos con el entorno Python. Para ello, crea un programa que automáticamente:

1. Lea el fichero proporcionado.
2. Cuente el número de líneas del fichero.
3. Cuente el número de veces que aparece el artículo «el» en el fichero (nota: elemento contiene los caracteres «el» pero no debe contar puesto que no contiene a el artículo «el».
4. Elimine las tildes del fichero y convierta todas las letras a minúscula.
5. Crea un fichero de texto con el nombre «resultado.txt» que contendrá en la primera línea el número de líneas del fichero, en la segunda línea el número de veces que aparece el artículo «el», y a partir de la tercera línea el texto original todo en minúsculas y sin tilde.

Sigue los siguientes pasos, empleando Pandas y Numpy tanto como sea posible:

- Carga en el entorno el «poblacionMunicipios.csv» proporcionado por el profesor respetando las tildes y caracteres españoles.
- Elimina las filas correspondientes a municipios sin población.
- Imprime el total de la población de todos los municipios.
- Crea un DataFrame de Pandas donde la primera columna sean todas las provincias que aparecen en el archivo, la segunda columna el número total de habitantes por provincia, la tercera la desviación típica en el número total de habitantes por provincia y la cuarta el número total de municipios por provincia.
- Carga el archivo «CP_Municipios.csv».
- Empleando el código de provincia y el código de municipio, cruza los dos datasets comentados hasta el momento. El resultado final debe ser un DataFrame.

- Genera un CSV llamado «faltan.csv» con todos los códigos postales que estén en «CP_Municipios.csv» pero que no tengan en «poblacionMunicipios.csv» **ningún municipio** con el que poder cruzar. Si todo cruza correctamente el archivo «faltan.csv» estará vacío. Se puede decir que «faltan.csv» son los códigos postales que se han perdido en el cruce a la hora de considerar la población.
- Agrupa la información por código postal. Se quiere un DataFrame con las siguientes columnas:
  - Código postal.
  - Número de municipios que tienen dicho código postal asignado.
  - Población: se calculará como la suma de la población de todos los municipios que incluyen a dicho código postal.
  - Provincia: a la que está asignada el código postal.

Se entregará un único notebook con el código y comentarios asociados.

Criterios de evaluación

- La información proporcionada es correcta
- El código se organiza correctamente mediante el empleo de funciones
- El código está documentado
- El código es eficiente

Test

1. Una librería de Python es:

A. Una función.

B. Una agrupación de funcionalidades.

C. Una aplicación.

D. Todas las anteriores.

2. NumPy es una librería útil para:

A. Procesar archivos CSV.

B. Visualización interactiva.

C. *Streaming*.

D. Cálculo matricial.

3. Pandas es:

A. Un lenguaje de programación.

B. Una librería de cálculo simbólico.

C. Una librería para facilitar el tratamiento de datos.

D. Todos los anteriores.

4. El proceso de trabajo de *data science*:

A. Comienza planteando una necesidad, oportunidad o problemática de negocio.

B. Comienza instalando la plataforma.

C. Comienza analizando los datos que tenemos.

D. Comienza generando unos modelos previos.

5. Para mostrar los valores concretos de las dimensiones de un vector con NumPy llamamos a:

   A. `shape()`.

   B. `size()`.

   C. `itemsize()`.

   D. No se puede.

6. Para llamar a la transpuesta de la matriz x con NumPy, ejecutamos:

   A. `x.Transpose()`.

   B. `x.Tps()`.

   C. `x.Transpose`.

   D. `x.T`.

7. Para obtener con Pandas un resumen rápido de la estructura e información contenida en un DataFrame llamamos a la función:

   A. `summary()`.

   B. `descr()`.

   C. `describe()`.

   D. `review()`.

8. La función `iloc` de panda:

   A. Nos permite acceder a los valores a través de la posición del índice.

   B. Nos devuelve la localización en memoria de un valor.

   C. Devuelve un puntero a memoria.

   D. Devuelve un valor lógico.

9. La sentencia df[df.A > 0]:

   A. No funciona.

   B. Hace que todos los valores de A en df sean mayores que cero.

   C. Cambia df eliminando todos los valores menores que cero.

   D. Muestra solo las filas de df que en su columna A tienen un valor mayor que cero.

10. Para hacer `join` con Pandas, empleamos:

A. `join()`.

B. `groupby()`.

C. `merge()`.

D. No se puede.