Llibreries de Python

Iloc:	Institut d'Ensenyaments a Distància de les Illes Balears	Imprès per:	Carlos Sanchez Recio
Curs:	Programació d'intel·ligència artificial	Data:	dilluns, 28 d'octubre 2024, 16:02
Llibre:	Llibreries de Python		

Descripció

Taula de continguts

1. Introducció

Una de les principals forces de Python és el gran conjunt de biblioteques que les comunitats de ciència de dades i d'aprenentatge automàtic han anat desenvolupant.

En aquest lliurament veurem les tres més importants:

- **NumPy** per al processament numèric
- **pandas** per a la manipulació i anàlisi de dades
- **Matplotlib** per a la visualització de dades

2. NumPy

NumPy, abreviatura de Numerical Python, és un dels paquets més importants per a la computació numèrica en Python. La majoria dels paquets computacionals que proporcionen funcionalitat científica treballen amb els ***ndarrays***, els arrays multimensionals de NumPy.

La informàtica orientada a matrius en Python es remunta a 1995, quan Jim Hugunin va crear la biblioteca numèrica. Durant els següents 10 anys, moltes comunitats de programació científica van començar a fer programació de matrius en Python, però l'ecosistema de la biblioteca s'havia fragmentat a principis dels anys 2000. El 2005, Travis Oliphant va poder forjar el projecte NumPy a partir dels llavors projectes Numeric i Numarray per reunir la comunitat al voltant d'un *framework* de matrius únic.

Aquestes són algunes de les característiques de NumPy:

- Arrays multidimensionals (*ndarrays*) eficients que ofereixen operacions aritmètiques ràpides orientades a matrius.
- Funcions matemàtiques per a operacions ràpides amb matrius de dades numèriques sense haver d'escriure bucles.
- Eines per llegir/escriure dades de matriu al disc i treballar amb fitxers assignats a memòria.
- Generació de nombres aleatoris
- Una API C per connectar NumPy amb biblioteques escrites en C, C++ o FORTRAN.

Com que NumPy proporciona una API C fàcil d'utilitzar, és senzill passar dades a biblioteques externes escrites en un llenguatge de baix nivell i també que les biblioteques externes retornin dades a Python com a matrius NumPy. Aquesta característica ha fet de Python un llenguatge preferit per embolicar bases de codi C/C++/Fortran heretades i donar-los un caràcter dinàmic i una interfície fàcil d'usar.

Tot i que NumPy proporciona una base computacional per al processament de dades numèriques generals, és més habitual utilitzar la biblioteca ***pandas*** com a base per a la majoria de les operacions estadístiques o analítiques, especialment amb dades alfanumèriques en forma de taula. En tot cas, tenir una comprensió de les matrius NumPy i la informàtica orientada a matrius ens ajudarà a utilitzar eines amb semàntica orientada a matrius, com ara *pandas*, de manera més eficaç.

IMPORTANT

Podeu trobar els exemples que anirem veient en aquest capítol de NumPy, amb més detalls, al **quadern Colab** que hem preparat.

Per ampliar la informació, podeu consultar la web de referència de NumPy.

2.1. Importació de NumPy

Per treballar amb la llibreria NumPy és recomanable emprar sempre la importació següent:

```
import numpy as np
```

Així, sempre que veiem ***np.*** en el codi, fa referència a NumPy.

També podríem importar la biblioteca a l'espai de noms global amb:

```
import numpy as *
```

Així no necessitaríem emprar el prefix *np*.

Però aquesta opció no és gens recomanable, ja que hi ha moltes funcions dins NumPy que tenen el mateix nom que les funcions predefinides en Python (com *min* i *max*).

2.2. Objecte vector multidimensional: ndarray de NumPy

Una de les característiques clau de NumPy és el seu objecte ***ndarray***, un ***array* multidimensional**, que serveix com a contenidor ràpid i flexible per a grans conjunts de dades en Python. Els *ndarrays* permeten realitzar operacions matemàtiques sobre blocs sencers de dades utilitzant una sintaxi similar a les operacions equivalents entre elements escalars.

Un *ndarray* conté **dades homogènies**; és a dir, tots els elements han de ser del mateix tipus. Cada *ndarray* té, entre d'altres, aquests atributs importants:

- ***ndim***: el número de dimensions
- ***shape***: una tupla que indica la mida de cada dimensió.
- ***dtype***: el tipus de dades dels elements de la matriu.

La manera més senzilla de crear un *ndarray* és fer-ho a partir d'un array de Python. Vegem, per exemple, com cream un ndarray a partir d'una matriu:

```
import numpy as np

a = [[1, 2, 3, 4],[5, 6, 7, 8]]
nda = np.array(a)
nda.ndim
```

Aquest fragment de codi retorna 2, ja que *nda* té dues dimensions (files i columnes. Amb l'atribut *shape* podem obtenir la mida de cada una d'aquestes dimensions

```
nda.shape
```

Retorna la tupla *(2, 4)* perquè tenim 2 files i 4 columnes. Tots els elements són de tipus *int*:

```
nda.dtype
```

```
dtype('int64')
```

ALERTA

La traducció d'*array* al català és **vector**. En aquests apunts i els seus quaderns de Collab utilitzarem indistintament els termes array i vector per referir-nos a un *ndarray*.

Vegem com crear un *ndarray* i omplir-lo de zeros (amb la funció *np.zeros*) i amb uns (amb *np.ones*):

```
zeros = np.zeros(10) # crea un ndarray de 10 posicions i l'omple amb zeros
uns = np.ones(5) # crea un ndarray de 5 posicions i l'omple amb uns
```

I també com omplir-lo amb els valors d'un rang, mitjançant la funció *np.arange*:

```
a1 = np.arange(5) # crea un ndarray amb els valors [0, 1, 2, 3, 4]
a2 = np.arange(3,7) # crea un ndarray amb els valors [3, 4, 5, 6]
a3 = np.arange(3,7,2) # crea un ndarray amb els valors [3, 5]
```

2.3. Vectorització: programació orientada als arrays

L'ús de matrius NumPy permet expressar molts tipus de tasques de processament de dades com a operacions de matrius compactes que, d'altra manera, podrien requerir bucles d'escriptura. Aquesta pràctica de reemplaçar bucles explícits per expressions de matriu es coneix habitualment com a **vectorització**.

IMPORTANT

En general, les operacions vectoritzades solen ser un o dos ordres de magnitud més ràpides que els seus equivalents purs de Python. Això té un gran impacte en el rendiment de qualsevol tipus de càlculs numèrics.

En el següent exemple vegem com podem operar un array i un escalar.

```
data = [ [1, 2, 3] , [4, 5, 6] ]
arr1 = np.array(data)
arr2 = 10 * arr1
arr2
```

En aquest cas, es multiplica per 10 cada un dels elements de l'array i es retorna un altre array:

```
array([[10, 20, 30],
       [40, 50, 60]])
```

També podem fer operacions entre dos arrays, on s'operaran posició a posició, tots els elements dels dos arrays. En el següent exemple, vegem com podem sumar dues matrius (de dues dimensions) de la mateixa mida.

```
arr3 = arr1 + arr2
arr3
```

El resultat és una nova matriu on en cada posició tenim la suma dels valors de les dues matrius en aquella posició:

```
array([[11, 22, 33],
       [44, 55, 66]])
```

ALERTA

Igual que amb la suma, si empram l'operador * multiplicarem, posició a posició, els valors de dues matrius.

```
arr3 = arr1 * arr2
arr3
```

```
array([[ 10,  40,  90],
       [160, 250, 360]])
```

Hem de tenir clar, però, que això no és l'operació algebraica de multiplicar dues matrius. Si volem fer la multiplicació de dues matrius, hauríem d'emprar la funció ***np.matmul***. Ho veurem amb més detall en l'apartat següent.

2.4. Funcions universals: operacions ràpides element a element

En l'apartat anterior hem vist com NumPy proporciona operadors vectoritzats que permeten treballar amb els elements de *ndarrays* de manera conjunta. De manera equivalent, NumPy també proporciona una sèrie de funcions per fer operacions amb *ndarrays* de manera vectoritzada, element a element. Són les anomenades **funcions universals**, o ***ufunc***. Això dona una solució més ràpida que programar els bucles per recórrer i operar tots els elements dels ndarrays que volem operar.

Per exemple, podem trobar un equivalent a l'operador vectoritzat *+* amb la *ufunc add*:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[10, 20, 30], [40, 50, 60]])
c = np.add(a,b)
c
```

Que dona com a resultat la suma, element a element, de les dues matrius:

```
array([[11, 22, 33],
       [44, 55, 66]])
```

IMPORTANT

En realitat, quan es fa servir l'operador ***+*** per sumar dues funcions, NumPy crida internament la *ufunc* ***add***.

No totes les *ufuncs* son binàries, és a dir, que tenen dos *ndarrays* com a arguments. També n'hi ha d'unàries, amb un únic argument. Per exemple, la *ufunc* ***negative*** canvia el signe (positiu o negatiu) de tots els elements d'un *ndarray*. Això és el mateix que multiplicar tots els elements del *ndarray* per -1.

```
d = np.array([[1, -2, 3], [-4, 5, -6]])
e = np.negative(d)
e
```

```
array([[-1,  2, -3],
       [ 4, -5,  6]])
```

IMPORTANT

Les *ufuncs* són objectes de la classe ***numpy.ufunc***, amb la qual cosa tenen una sèrie d'atributs i mètodes, tot i que no entrarem en més detalls. Podeu trobar-ne més detalls en la documentació de Python.

A https://numpy.org/doc/stable/reference/ufuncs.html podeu trobar la llista de totes les *ufuncs* que proporciona NumPy. N'hi ha de diversos tipus:

- Operacions matemàtiques i d'àlgebra lineal, com *add* i *negative* que hem vist. Són molt utilitzades en l'anàlisi de dades i per molts algorismes d'aprenentatge automàtic

- Operacions trigonomètriques
- Operacions amb nombres enters a nivell de bit
- Operacions amb nombres reals
- Operacions de comparació

Ens aturam un moment en aquestes darreres, les operacions de comparació. Podem trobar, per exemple, una funció ***greater***, que compara element a element dos *ndarrays* i per a cada una d'aquestes comparació retorna un booleà: *True* si l'element del primer *ndarray* és major que el del segon, i *False* altrament.

```
f = np.array([[1, 9, 5], [3, 8, 7]])
g = np.array([[6, 2, 4], [1, 9, 5]])
h = np.greater(f,g)
h
```

```
array([[False,  True,  True],
       [ True, False,  True]])
```

Val la pena mencionar també les funcions de comparació ***maximum*** i ***minimum***, que comparen element a element dos *ndarrays* i per a cada una d'aquestes comparacions retornen el més gran (*maximum*) o el més petit (*minimum*) dels dos valors.

```
h = np.maximum(f,g)
h
```

```
array([[6, 9, 5],
       [3, 9, 7]])
```

No hem de confondre les *ufuncs maximum* i *minimum* de NumPy amb les funcions *max* i *min* de Python, que no són vectoritzades. Aquestes dues darreres comparen els dos arguments en global i retorna quin dels dos és més gran.

```
f = [[1, 9, 5], [3, 8, 7]]
g = [[6, 2, 4], [1, 9, 5]]
h = max(f,g)
h
```

Retorna que l'element més gran és la llista *g*:

```
[[6, 2, 4], [1, 9, 5]]
```

Tampoc no hem de confondre les *ufuncs maximum* i *minimum* amb les funcions *max* i *min* de Numpy, que retornen l'element màxim i mínim d'un ndarray, o d'un dels seus eixos.

```
f = [[1, 9, 5], [3, 8, 7]]
maxim = np.max(f)
maxim
```

Si volem recuperar tota la fila del màxim, seria així:

```
filamaxim = np.max(f, axis=0)
filamaxim
```

Que ens retorna:

```
array([3, 9, 7])
```

I si volem tota la columna:

```
colmaxim = np.max(f, axis=1)
colmaxim
```

Que ens retorna:

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

Multiplicació de matrius

A diferència de la suma, l'operació algebraica de multiplicar dues matrius **no** és el producte, element a element, de les dues matrius. És una operació una mica més complexa.

Vegem-ho amb un exemple:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 1x5+2x7 & 1x6+2x8 \\ 3x5+4x7 & 3x6+4x8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

La primera posició de la matriu obtinguda és el resultat d'agafar els elements de la primera fila de la primera matriu i els de la primera columna de la segona matriu, els multiplicam un a un i sumam tots els resultats. Per a la primera fila segona columna de la matriu resultat feim el mateix amb la primera fila de la primera matriu i la segona columna de la segona. I així successivament.

La única restricció és que el número de columnes de la primera matriu ha de ser igual al número de files de la segona matriu.

Vegem una implementació no vectoritzada amb Python, que potser ens ajuda a entendre millor el funcionament de l'algorisme:

```
a = [[1,2],[3,4]]
b = [[5,6],[7,8]]
files_a = len(a)
files_b = len(b)
columnes_a = len(a[0])
columnes_b = len(b[0])
assert columnes_a == files_b, '''El número de columnes
 de la matriu a
 ha de ser igual que
 el número de columnes
 de la matriu b'''

# Omplim la matriu resultat (c) amb None
c = []
for i in range(files_b):
    c.append([])
    for j in range(columnes_b):
        c[i].append(None)

# Calculam els valors amb un triple bucle
for k in range(columnes_b):
    for i in range(files_a):
        suma = 0
        for j in range(columnes_a):
            suma += a[i][j]*b[j][k]
        c[i][k] = suma

c
```

El resultat és la següent llista:

```
[[19, 22], [43, 50]]
```

NumPy ens proporciona una *ufunc* per fer el producte de dos *ndarrays*, no només de dues dimensions sinó de qualsevol número: ***matmul***:

```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
c = np.matmul(a,b)
c
```

El resultat és un *ndarray*:

```
array([[19, 22],
       [43, 50]])
```

Una altra *ufunc* que permet multiplicar dos *ndarrays* és ***dot***.

```
c = np.dot(a,b)
c
```

```
array([[19, 22],
       [43, 50]])
```

No obstant això, la documentació de NumPy recomana emprar *dot* només per a *ndarrays* d'una dimensió i *matmul* per a dimensions majors que 1.

D'altra banda, des de Python 3.5 s'ha afegit un nou operador en el propi llenguatge per multiplicar matrius: @. És l'equivalent de la *ufunc matmul*.

```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
c = a @ b
c
```

```
array([[19, 22],
       [43, 50]])
```

L'operació de multiplicar dues matrius és molt habitual en diversos àmbits de la informàtica, especialment en el processament d'imatges, ja que una imatge és una matriu de píxels. Tant és així que DeepMind (una companyia de Google especialitzada en aprenentatge automàtic basat en xarxes neuronals profundes i que ha desenvolupat AlphaZero, un programa que aprèn tot sol en unes poques hores a jugar a escacs i go a un nivell superior als campions del món humans) està treballant en AlphaTensor, una evolució d'AlphaZero dedicat a trobar algorismes que multipliquin matrius d'una manera més eficient.

Podeu trobar més detalls sobre com funciona AlphaTensor a diverses notícies d'alguns mitjans, com per exemple a Xataka. I si voleu aprofundir encara més, us recoman que mireu l'article que varen publicar els investigadors de DeepMind a la revista Nature a l'octubre de 2022. Per cert, un dels firmants de l'article és el recent Premi Nobel, Demis Hassabis.

2.5. Entrada/sortida de fitxers amb arrays

NumPy és capaç de desar i carregar dades al disc i des del disc en format de text o binari.

Només mencionarem el format binari integrat de NumPy, ja que sovint es prefereixen les funcionalitats que proporciona la llibreria pandas, així com altres eines, per carregar dades de text o tabulars.

np.save i ***np.load*** són les dues funcions fonamentals per desar i carregar de manera eficient les dades de la matriu al disc. Les matrius es guarden de manera predeterminada en un format binari sense comprimir, amb l'extensió de fitxer **.npy**.

2.6. Generació de nombres aleatoris

El mòdul ***numpy.random*** complementa el mòdul *random* predefinit a Python integrat amb funcions per generar de manera eficient matrius senceres de valors de mostra aleatoris, a partir de molts tipus de distribucions de probabilitat. En el lliurament 2 del mòdul de Sistemes de Big Data es veu què són les distribucions de probabilitat. Aquí, simplement ens podem quedar amb que determinen la manera en què es van generant les seqüències de nombres aleatoris.

Dues de les més habituals són la distribució uniforme (***numpy.random.rand***) i la distribució normal (o gaussiana) estàndard (***numpy.random.randn***). La primera genera un nombre real aleatori (amb decimals) en l'interval [0,1), és a dir, incloent el 0, però no l'1. En aquest cas, tots els valors tenen la mateixa probabilitat d'aparèixer. En canvi, en una distribució normal estàndard (amb mitjana 0 i desviació estàndard 1), és més probable que es generi un nombre real proper a 0. Aproximadament un 70% dels valors generats estarà entre -1 i +1 i només un 0,006% dels valors estaran fora de l'interval entre -4 i +4.

Vegem un exemple que fa servir *np.random.randn* per generar un *ndarray* de dues dimensions, amb 2 files i 3 columnes:

```
arr = np.random.randn(2,3)
arr
```

El resultat és:

```
array([[-0.95101158, -0.73277403, -0.15739885],
       [ 0.11041919, -0.09447163,  0.572265  ]])
```

Podeu consultar més detalls sobre la generació de nombres aleatoris en la documentació oficial de NumPy: https://numpy.org/doc/stable/reference/random/index.html

2.7. Indexació i selecció

Hi ha moltes formes en què podem voler triar un subconjunt d'un ndarray. Per això la indexació i selecció d'elements dins d'un ndarray és un tema complex.

Vegem un exemple d'indexació amb un vector unidimensional:

```
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
a[5]
```

Retorna el valor de la posició 5 (començant per 0): 6.

Vegem ara un altre exemple de selecció (*slicing*):

```
a[2:4]
```

Retorna els valors de les posicions 2 i 3:

```
array([3, 4])
```

En el cas de ndarrays de més dimensions és quan apareixen més opcions. Vegem un primer exemple amb una matriu:

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

Retorna tota la fila amb índex 2 (la tercera, perquè comencam per 0:

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

Podeu continuar ara amb els apartats d'Indexació i selecció, Indexació amb seleccions i Indexació booleana del quadern de Colab de Numpy, on trobareu molts més detalls.

3. pandas

La biblioteca **pandas** és una eina que s'emprarà àmpliament durant el curs. Conté estructures de dades i eines de manipulació de dades dissenyades per fer que la neteja i l'anàlisi de dades amb Python siguin ràpides i fàcils. el nom de pandas ve de "Python data analysis".

pandas està construida sobre la biblioteca NumPy i, s'utilitza sovint en conjunt amb altres biblioteques numèriques com SciPy, biblioteques analítiques com statsmodels i scikit-learn i biblioteques de visualització de dades com matplotlib. pandas adopta parts importants de l'estil de la computació basada en matrius de NumPy, especialment funcions basades en matrius i una preferència pel processament de dades sense bucles *for*.

La diferència més important amb NumPy és que pandas està dissenyat per treballar amb dades tabulars i heterogènies, com podrien ser les d'un full de càlcul o una taula d'una base de dades relacional. NumPy, per contra, és més adequat per treballar amb dades de matriu numèrica homogènia.

pandas és una biblioteca de programari lliure, amb una gran comunitat de desenvolupadors, que ha crescut fins a arribar a més de 2.000 col·laboradors.

Se sol fer servir la següent convenció d'importació per a pandas.

```
import pandas as pd
```

Així, sempre que veiem **pd.** en el codi, fa referència a pandas.

També és còmode importar **Series** i **DataFrame** a l'espai de noms local, ja que s'utilitzen amb molta freqüència.

```
from pandas import Series, DataFrame
```

3.1. Estructures de dades

Per començar a treballar amb pandas, cal familiaritzar-se amb les seves dues estructures de dades fonamentals: Series i DataFrame. Tot i que no resolen qualsevol problema, donen una base sòlida i fàcil d'usar per a la majoria d'aplicacions.

Series

Una sèrie (*Series*) és un objecte vectorial unidimensional que conté una seqüència de valors (de tipus semblants als tipus NumPy) i un vector associat d'etiquetes de dades, anomenat *index*. La Series més simple es forma a partir d'un vector de dades.

Ho veurem en detall en el **quadern de Colab sobre Series de pandas**.

DataFrame

Un DataFrame representa una taula rectangular de dades i conté una col·lecció ordenada de columnes, que poden ser de diferents tipus (numèric, cadena de caràcters, booleà, etc.). El DataFrame té un índex de filera i un altre de columna. Podem pensar-hi com un diccionari (*dict*) de Series que totes comparteixen el mateix índex. Internament, les dades s'emmagatzemen com un o més blocs bidimensionals més que no com a llista, diccionari o una altra col·lecció de vectors unidimensionals.

Tot i que un DataFrame és físicament bidimensional, es pot fer servir per representar dades en moltes dimensions utilitzant indexació jeràrquica. Això és un element important d'algunes tècniques avançades de gestió de dades en pandas.

Hi ha moltes formes de construir un DataFrame; una de les més freqüents és a partir d'un diccionari de llistes de la mateixa longitud, o de vectors NumPy.

Trobareu els detalls al **quadern de Colab sobre DataFrame de pandas**.

3.2. Operacions d'anàlisi de dades

La llibreria pandas, i molt especialment, l'estructura *DataFrame*, és molt àmpliament utilitzada en l'àmbit de l'anàlisi de dades.

L'estructura en forma de taula d'un dataset o conjunt de dades, on tenim diverses columnes de diferents tipus i una fila per a cada instància, es representa perfectament mitjançant un *DataFrame*. En altres situacions, com poden ser sèries temporals d'una variable, també és habitual fer feina amb *Series*.

pandas ofereix una sèrie de funcions útils per a l'anàlisi de dades, tant per a *Series* com per a *DataFrame*.

Ho veurem en detall en el **quadern de Colab sobre operacions d'anàlisi amb pandas**.

Podeu trobar molta més informació a la documentació de l'API de Python:

- Operacions de lectura i escriptura de dades
- Atributs i mètodes de l'Objecte Series
- Atributs i mètodes de l'Objecte DataFrame

4. Matplotlib

Realitzar visualitzacions informatives (de vegades anomenades *plots*) és una de les tasques més importants de l'anàlisi de dades. Pot ser una part del procés d'exploració, per exemple, per identificar outliers o transformacions convenients per a les dades, o una forma de generar idees per a models. En altres casos, construir una visualització interactiva per a la web pot ser l'objectiu final.

Python té moltes llibreries afegides per fer visualitzacions dinàmiques. Ens centrarem en matplotlib i les llibreries que s'hi basen.

matplotlib és un paquet de visualització per crear gràfics d'alta qualitat, principalment bidimensionals. El projecte el començà John Hunter el 2002 per aconseguir des de Python una interfície de gràfics com la de MATLAB. Les comunitats de matplotlib i IPython han col·laborat per simplificar els gràfics interactius des de la consola IPython (ara els quaderns Jupyter). matplotlib té suport per a diversos backends d'interfície gràfica d'usuari en tots els sistemes operatius i pot exportar visualitzacions a tots els formats habituals de mapa de bits o vectorials (PDF, SVG, JPG, PNG, BMP, GIF, etc.).

Amb el temps, matplotlib ha fet possible que es desenvolupin un nombre de toolkits de visualització de dades que l'utilitzen. Un dels més utilitzats és **seaborn**.

En aquest **quadern de Colab sobre matplotlib** teniu diverses funcions i opcions que us seran útils per representar gràfics en anàlisi de dades i aprenentatge automàtic.

5. Altres biblioteques

NumPy, pandas i Matplotlib són probablement les tres biblioteques de Python més populars entre les comunitats d'intel·ligència artificial i ciència de dades. Però, al voltant d'aquestes s'ha anat desenvolupant un gran ventall d'eines amb Python.

En el camp del processament matemàtic, a més de NumPy, podem trobar **SciPy**, una biblioteca molt potent, que proporciona algorismes per a molts de problemes de ciència i enginyeria com optimització, integració, interpolació, problemes de valors propis, equacions algebraiques, equacions diferencials i estadístiques. De fet, SciPy pretén competir amb MATLAB o GNU Octave.

En l'àmbit de l'anàlisi de dades, a més de pandas, podem mencionar també **statsmodels**, un mòdul que proporciona classes i funcions per a l'estimació de molts models estadístics, així com capacitat per dur a terme proves estadístiques i exploració de dades estadístiques. També val la pena destacar les API d'**Apache Spark**, un framework per a l'anàlisi de grans conjunts de dades de forma distribuïda, que veurem en els mòduls de big data.

Pel que fa a la visualització de dades, ja hem mencionat **seaborn**, una biblioteca construida sobre matplotlib, més sofisticada i orientada a generar gràfics més elaborats i atractius, d'una manera més senzilla. Se la considera un superconjunt de matplotlib. Una altra biblioteca per a la visualització de dades és **Datashader**, que està orientada a millorar l'eficiència per treballar amb grans conjunts de dades.

En l'àmbit de l'aprenentatge automàtic, el principal referent és la biblioteca **scikit-learn** , que veurem en gran detall en el mòdul de Sistemes d'aprenentatge automàtic. Combina una àmplia selecció de mètodes d'aprenentatge automàtic supervisat i no supervisat, amb eines per a la selecció i avaluació dels models, la càrrega i la transformació de dades i la persistència dels models. Aquests models es fan servir en classificació, agrupació (*clustering*), predicció i d'altres tasques habituals en aprenentatge automàtic.

scikit-learn està construït sobre NumPy, SciPy i matplotlib. De fet, el seu nom prové de la noció que és un *SciKit* (un *toolkit* de SciPy), una extensió desenvolupada i distribuïda de forma separada de SciPy.

Si ens centram en l'aprenentatge automàtic basat en xarxes neuronals profundes, també tenim múltiples eines per treballar amb Python: **TensorFlow**, **PyTorch** o **Keras** en són alguns exemples.