CE888: Data Science and Decision Making

Lab 1: Setting up and Numpy Practice

Ana Matran-Fernandez

14 January 2019

Institute for Analytics and Data Science
University of Essex

Table of contents

Setting up

The Linux Virtual Machine

Normally we start the first lab downloading and installing the LVM.

However, this year has been a bit tricky, so we'll work on the local lab machine directly for the next few weeks.

I will upload instructions on how to get the LVM to work for those who want to try.

Creating a Github account and CE888 repository

You need to do this so that we can check your lab practice

- ☐ Go to `http://www.github.com`
- ☐ Create an account (if you don’t have one already)
- ☐ You can request a student account to get private repos if you want
- ☐ Create a new **public** repository called `ce888labs`
- ☐ Set the .gitignore to Python
- ☐ Add a default README.md
- ☐ Make sure you are in your home directory and clone your git repo: `git clone <reponame>`

Downloading the lab scripts

- ☐ Go to the Moodle page for this week:
- ☐ `https://moodle.essex.ac.uk/course/view.php?id=6683&section=7`
- ☐ Download the notebook for today's practice into your local github lab directory (e.g., `/labs/lab1`).
- ☐ Commit and push the file:
  - ☐ Go to the folder
  - ☐ `git add -A -v`
  - ☐ `git commit -m "Message"` (Write your own message!!
  - ☐ `git push origin master`

Lab exercise 1

Now you will work on the first part of the JuPyter notebook.

- ☐ Start JuPyter notebooks by typing `jupyter notebook` from the `/labs/lab1/` folder.
- ☐ This will open a browser in your computer.
- ☐ Navigate to and open the `notebook1.ipynb` file.
- ☐ Using the emotion detector, print the emotional content of each message (a vector of six numbers)
- ☐ Once you are done, save your changes in github:
  - ☐ Go inside your lab directory
  - ☐ `git add -A -v`
  - ☐ `git commit -m "Exercise 1 of lab1"` (or something similar)
  - ☐ `git push origin master`

The Scipy ecosystem

Not from scratch!

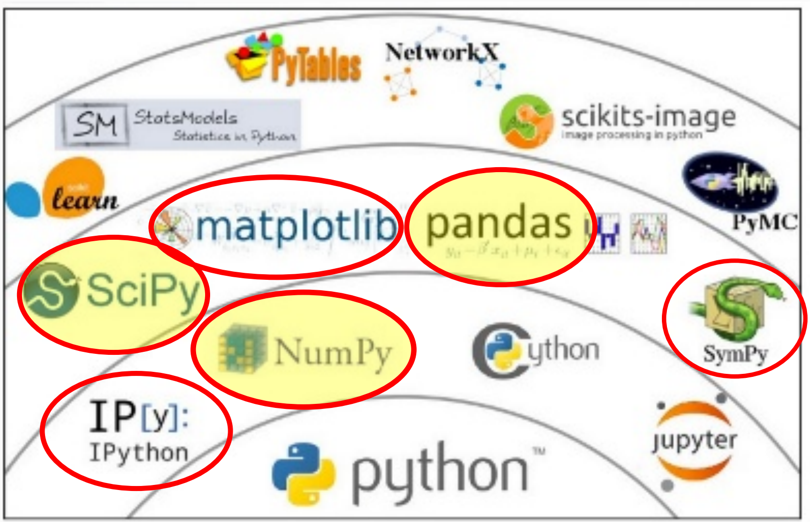
Hopefully you’ve realised that some of the things that you had to do in the first part of the lab were not that easy (e.g., matrix multiplication).

Implementing things “from scratch” is great for understanding how they work. But it’s generally not great for performance (unless you’re implementing them specifically with performance in mind), ease of use, rapid prototyping, or error handling. We’ll see this later!

In practice, you’ll want to use well-designed libraries that solidly implement the fundamentals.

Now we’ll see a few of those that are great for scientific purposes and data analysis.

The Scientific Python Ecosystem (I)



The Scientific Python Ecosystem (II)

The SciPy *ecosystem* is a collection of open source software for scientific computing in Python.

Core packages:

- NumPy, the fundamental package for numerical computation. Creates and uses arrays, matrices and mathematical operators.
- SciPy library, a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.
- Matplotlib, plotting package, basis for every advanced plotting libraries, e.g., Seaborn.

The Scientific Python Ecosystem (III)

High-level libraries:

- SciKit-Learn, machine learning tools and algorithms.
- Pandas data analysis library. Provides structured data objects that make data-analysis of large datasets easier.
- PyTables to access data stored in the HDF5 format, minimises memory usage.
- StatsModels classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration.

NumPy

Why NumPy?

Short for Numerical Python, NumPy is the fundamental package required for high performance scientific computing and data analysis in the Python ecosystem.

It's the foundation on which nearly all of the higher-level tools, such as Pandas and scikit-learn, are built.

Many NumPy operations are implemented in C, making them super fast. For data science and modern machine learning tasks, this is an invaluable advantage.

NumPy ndarray

NumPy provides an extension to the Python types, the `ndarray`, that is much more efficient at storing and manipulating numerical data than the built-in Python types.

ndarrays are n-dimensional arrays of homogeneous data types: this allows lower memory use and faster operations.

They have fixed sizes: every modification to the shape results in a new array being stored.

Importing NumPy and creating an array

```
import numpy as np  # importing convention

a = np.array([1, 2, 3, 4, 5])
a.ndim  # number of dimensions of the array (1)
a.shape  # shape of the array (5,)

b = np.array([[3, 4, 5], [6, 7, 8]])
b.ndim  # 2
b.shape  # (2, 3)

# Some special arrays
np.zeros(5)  # 1-D array of 5 zeros
np.ones((6, 2, 3))  # 3-D array of one's
np.empty((2, 7))  # 2-D empty array
np.eye(4)  # 4-by-4 identity matrix
```

Indexing

There are multiple ways of selecting specific rows and columns from a NumPy array.

The easiest way is slicing:

```
myArray[0]  # select first row
myArray[3, 5:7]  # specific elements from the 4th row
myArray[:2, 4, 26:30]
# and so on
```

What if we don't know which items exactly we want to access?

We may want to retrieve slices of the array based on a condition. We can do this using Boolean indexing.

Boolean indexing

We can apply Boolean operators (which you saw in the lecture yesterday) to an ndarray to obtain a binary mask:
`mask = myArray != 0` returns a boolean array of the same shape of myArray, telling us all the elements that are not 0.

We can then do `myArray[mask]` to see only those values that were different from 0.

The `and` and `or` operators don't work with ndarrays. Instead, use & and |, respectively.

The results of these operations can be aggregated into a single value with the `any()` and `all()` methods.

We can also use the `np.where()` method to access specific values:

```
mask = np.where(myArray < 0)
myArray[mask] = 0
```

Changing the Shape of the Array

We have seen how to visualise and select data from an existing ndarray, how can we add/delete rows/columns to it?

```
# Assume we have arrays of shapes:
array1.shape  # (20, 4)
array2.shape  # (10, 4)
array3.shape  # (20, 87)
np.hstack((array1, array3))  # stacks arrays horizontally; (20, 91)
np.vstack((array1, array2))  # stacks arrays vertically; (30, 4)
np.append(array1, array2)  # flattens both arrays and appends array2
                           # to the end of array1; (120,)
np.append(array1, array2, axis=0)  # (30, 4)

# Remove a row/column:
np.delete(array1, 2, axis=0)  # removes the 3rd row; (19, 4)
np.delete(array1, 2, axis=1)  # removes the 3rd column; (20, 3)
```

Transposing and Reshaping (I)

Transposing an array: `myArray.T` or `myArray.transpose()`

If myArray.ndim > 2, we can also permute its dimensions using transpose

```
# Assume we have:
array2d.shape #(20, 87)
array3d.shape #(10, 3, 28)

array2d.T  # (87, 20)
array3d.T  # (28, 3, 10)
array3d.transpose()  # (28, 3, 10)
array3d.transpose(1, 0, 2) # (3, 10, 28)
```

Transposing and Reshaping (II)

We can reshape our array:

```
# Assume we have:
array2d.shape #(20, 87)
array3d.shape #(10, 3, 28)

array2d.reshape((4, -1))  # (4, 435)
# -1 means we don't need to write this dimension;
# it's calculated automatically for us

# we can put the -1 anywhere
array2d.reshape((-1, 30)) # (58, 30)
# convert to 3D array
array2d.reshape((10, -1, 3))  # (10, 58, 3)

array3d.reshape((30, -1, 28))  # (30, 1, 28)
```

Common operations

The power and beauty of NumPy are in its ability to vectorize operations. This means that we don't need to calculate values 1-by-1 using a for loop. E.g., `np.sqrt(myArray)`.

We can also obtain some summaries of the array:

```
myArray.mean() # average across the whole array
myArray.mean(axis=0) # average across columns
myArray.sum(axis=1) # aggregates all values across rows
myArray.std(axis=2) # standard deviation across the 3rd dimension

# How many positive values
(myArray > 0).sum()

# Sorting values in place
myArray.sort()
# Keep the array as it is and return a sorted copy:
np.sort(myArray)
```

NumPy as a Random Number Generator

The module `numpy.random` includes functions that we can use to generate random numbers.

```
# Array of shape (10, 23) of random numbers between [0, 1)
np.random.rand(10, 23)
# Array of 130 random numbers between [0, 50)
50 * np.random.rand(130)
# Random integers in [min, max)
np.random.randint(min, max)

# We can also draw numbers from a non-uniform distribution:
np.random.poisson(2.0)  # Poisson distribution with lambda=2
np.random.normal(1, 2, size=20)  # 20 values from a
#normal distribution with mean 0 and std 2

# If we set a seed, the numbers generated will be the same every time.
# Very important for debugging!!
np.random.seed(32)
# But very dangerous!!!!!!!!!!!!
```

Generating ranges

Two main functions in NumPy can be used to generate linear ranges of values:

- `np.arange(start, stop, step)` generates values between [start, stop), with a `step` between consecutive values (by default, 1).
- `np.linspace(start, stop, num, endpoint=True)` returns `num` evenly spaced numbers over the specified interval. It's the preferred method when the step is not an integer.

These are very useful, for example, to create a time vector for plotting.

Lab exercise 2

- ☐ Go back to your JuPyter notebook.
- ☐ Attempt Part 2.
- ☐ Once you are done, save your changes in Github:
  - ☐ Go inside your lab directory
  - ☐ `git add -A -v`
  - ☐ `git commit -m "Message"`
  - ☐ `git push origin master`
- ☐ Call me or the teaching assistant to show us your work!
- ☐ This is very important!! Otherwise you won't get the marks for the lab!