Python Applied to Machine Learning and Statistics

Lecture 02: Numpy and Data Visualization

Ricardo Cruz   Pedro Costa   Kelwin Fernandes

September 14, 2016

General Notes

As mentioned in the previous lecture:

- Python is a general purpose language
    - It is used in everything from user interfaces to web servers
    - The scientific computing community is strong, but is less centralized than Matlab
    - (But not as decentralized and messy as R :P)
    - In this first lecture, we will cover only language basics, not the packages for scientific computing

Numpy

The *de-facto* package for linear algebra is numpy.

Usually, it is shortended to np:

```
import numpy as np
```

Numpy features two main structures:

- np.ndarray — array of multiple dimensions
- np.matrix — matrix (two dimensions)

> Why ever use np.matrix when np.ndarray supports two dimensions?

I don't know. I seldom see np.matrix used.

- np.ndarray supports everything np.matrix does
- np.matrix is slightly more convenient when working with matrices because it supports operators such as * and **
- We are going to only use np.ndarray

Numpy Example

Open a CSV file and add some random noise.

```python
import numpy as np
X = np.loadtxt(’file.csv’, [np.float64, np.float64, np.int8],
               delimiter=’,’, skiprows=1)
y = X[:, -1]
X = X[:, :-1]
X += np.random.randn(X.shape[0], X.shape[1]) * [4, 1]
```

By default numpy always works with `np.float64`. But you may want to change `dtype` when opening a CSV file or creating a vector in order to use less memory. The following types are available:

`np.int`	8, 16, 32 and 64 bits
`np.uint`	8, 16, 32 and 64 bits
`np.float`	16, 32 and 64 bits
`np.complex`	64 and 128 bits

Complex numbers are represented using two 32-bits or 64-bits floats.

Numpy GPU

Notes on using Numpy with the GPU:

Numpy does not support computations in the GPU.

GPU is supported by such packages as Theano and TensorFlow (for deep learning).

However, keep in mind that:

By default numpy always works with `np.float64`.

Some GPUs work in 32-bits. Therefore, you may want to use `dtype=np.float32` in such cases (to avoid conversion and memory waste).

Numpy vs MATLAB

Summary of **Numpy for MATLAB users** [1] [2]

MATLAB	Numpy
`size(a)`	`a.shape`
`a(1:5,:)`	`a[0:5,:]` or `a[0:5]` or `a[:5]`
`a(end-4:end,:)`	`a[-5:]`
`a.’`	`a.transpose()` or `a.T`
`a * b`	`a.dot(b)`
`a .* b`	`a * b`

The main conceptual difference is that Numpy supports **arithmetic broadcasting.** That is, you can do the following element-wise multiplication: (6,3) * (3). It automatically assumes you want to multiply by column. In MATLAB, you would have to use `bsxfun(@times,r,A)` or first use `repmat()`.

[1] https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

[2] http://mathesaurus.sourceforge.net/matlab-numpy.html

Numpy Exercise

1. Create a random vector called a of size 8 by sampling the Gaussian distribution $\mathcal{N}(10, 2)$.

APIs: `numpy.randn`, `numpy.random.normal`, `scipy.stats.normal`

```
a = np.random.normal(10, 2, 8)
```

2. Create another random vector called b of size 8 by sampling the Pareto distribution $\mathcal{P}(1)$.

APIs: `numpy.random.pareto`, `scipy.stats.pareto`

```
b = np.random.pareto(1, 8)
```

3. Create a matrix (8, 2) called A by merging the previous two vectors.

APIs: `numpy.vstack`, `np.c_`

Suggestions:

```
A = np.vstack((a, b)).T
A = np.c_[a, b]
A = np.concatenate(([a], [b]), 0).T
A = np.concatenate((a, b)).reshape((8, 2))
```

Numpy Exercise

5. Create $B_{ij} = \begin{cases} 5, & \text{if } A_{ij} > 1 \\ 0, & \text{otherwise.} \end{cases}$

Suggestions:

```
B = np.asarray(
  [[5 if aij > 1 else 0 for aij in ai] for ai in A])
```

```
B = A.copy()
B[A > 1] = 5
B[A <= 1] = 0
```

```
B = np.zeros(A.shape)
B[A > 1] = 5
```

```
B = (A > 1) * 5
```

It is probably a good idea to finish this off with:

```
B = B.astype(np.int8)
```

to save memory!

Numpy Load

Previously we have used `np.loadtxt` (with `delimiter=','`) to read from a CSV file.

But numpy has another function to open files: `np.genfromtxt`.

```
numpy.loadtxt(fname, dtype=<type 'float'>, comments='#',
    delimiter=None, converters=None, skiprows=0, usecols=None,
    unpack=False, ndmin=0)
numpy.genfromtxt(fname, dtype=<type 'float'>, comments='#',
    delimiter=None, skip_header=0, skip_footer=0, converters=None
    , missing_values=None, filling_values=None, usecols=None,
    names=None, excludelist=None, deletechars=None, replace_space
    ='_', autostrip=False, case_sensitive=True, defaultfmt='f\%i'
    , unpack=None, usemask=False, loose=True, invalid_raise=True,
     max_rows=None)
```

`np.genfromtxt` has more stuff such as for `NaN` handling. I don't know why, but `numpy.loadtxt` seems to be more often used.

Functions

In the last lecture, we have seen that Python uses pass-by-object-reference.

What happens then if

```python
def fn(b):
    b += 1
a = np.asarray(2)
fn(a)
print(a)
```

What is the value of a?

a = 3

Numpy vs Python Lists

Python lists can always be converted into a np.ndarray:

You can always painless convert a python list to np.ndarray using:

```
np.asarray([0, 1, 2, 3, 4])
np.asarray(range(5))  # also works for generators!
```

np.asarray(v) = np.array(v, copy=False)

so always use np.asarray(v) to avoid copying memory (unless of course you want to copy the memory).

Avoid Python Lists

In general, I would avoid Python lists.

Let us say I want to create a (nrow,ncol) matrix.

```
ncol = 2
nrow = 10
m = [[0 for _ in range(nrow)] for _ in range(ncol)]
```

Use a numpy `ndarray`:

```
m = np.zeros((ncol, nrow), int)
```

Numpy is incredibly efficient. It does very little copying, even when indexing. Internally, it works using the concept of views, and performs implicit copy on modification.

Furthermore, Python list elements may be discontinuous in memory. Numpy arrays are contiguous in memory. Making them much faster to access.

Scipy

A popular extension to Numpy is **Scipy.**

Scipy adds:

- popular statistical distribution and hypothesis testing
- linear algebra, signal processing and fourier transformations
- numerical methods, like integration and optimization

Pandas

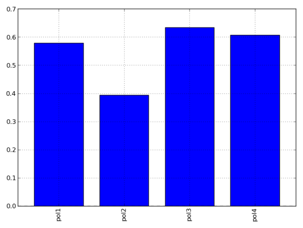
Pandas

If you are used to R `data.frame` then you are going to miss things like accessing columns by column name.

There is a widely used data-frame package for Python called pandas.

(This package is particularly useful when dealing with time series because it supports a lot of timeseries functionality we are not going to cover here.)

```
import pandas as pd
df = pd.read_csv('file.csv')
print(df['col1'].mean())
df.mean(axis=0).plot(kind='bar')
```



Exercise: open a CSV file and print `DataFrame.describe()`.

Pandas: indexing

```
import numpy as np
import pandas as pd
df = pd.DataFrame({'age': np.random.randn(10)*10+50,
                   'height': np.random.randn(10)*2+160,
                   'gender': ['male' if i == 0 else 'female'
                              for i in np.random.randint(0, 2, 10)]})
print(df.ix[2])  # age 51.142, gender female, height 159.245
```

In this case `df.ix` = `df.loc` = `df.iloc`. But you can also label your rows such as is the case in timeseries, in which case these functions would not be equivalent.

Pandas also supports some data access offered in R packages like `dplyr`.

```
df.groupby('gender').mean()
#               age      height
# gender
# female  53.870850  158.201459
# male    53.491803  158.971695
```

You can also do arithmetic with Pandas `DataFrame` and `Series`.

Pandas and Numpy

To convert to Numpy:

```
a = df.as_matrix()
```

Even though the function is called `DataFrame.as_matrix()`, it returns a `numpy.ndarray`, not a `numpy.matrix`. (The `numpy.matrix` structure is rarely used.)

Pandas is also faster reading CSV files than numpy[3]. Furthermore, it offers more options when loading CSV files.

[3] http://stackoverflow.com/questions/18259393/numpy-loading-csv-too-slow-compared-to-matlab

Matplotlib

Matplotlib

The *de-facto* package for plotting graphics in python is matplotlib.

Matplotlib contains an API[4] called `pyplot` that is inspired in MATLAB.

See: `http://matplotlib.org/1.4.3/api/pyplot_api.html`

This (I think) is the most widely used API, and is the API we are going to use.

[4] API = Application Programmer's Interface

Matplotlib

Example of synthetically created data:

```
from sklearn.datasets import make_blobs
X, y = make_blobs()
print(X[:3])
```

```
[[ 6.47751451  2.68612786]
 [ 7.29345147  4.14041813]
 [-9.29366385 -9.55913478]]
```

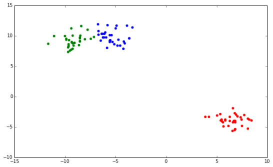
```
print(y[:3])
```

```
[0 0 2]
```

Matplotlib

Example of synthetically created data:

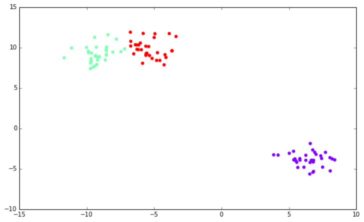
```python
from sklearn.datasets import make_blobs
X, y = make_blobs(centers=3)

from matplotlib.pyplot as plt
plt.scatter(X[y == 0, 0], X[y == 0, 1], color='red')
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='green')
plt.scatter(X[y == 2, 0], X[y == 2, 1], color='blue')
plt.show()

# OR
colors = ['red', 'green', 'blue']
plt.scatter(X[:, 0], X[:, 1], color=[colors[_y] for _y in y])
plt.show()
```



Matplotlib

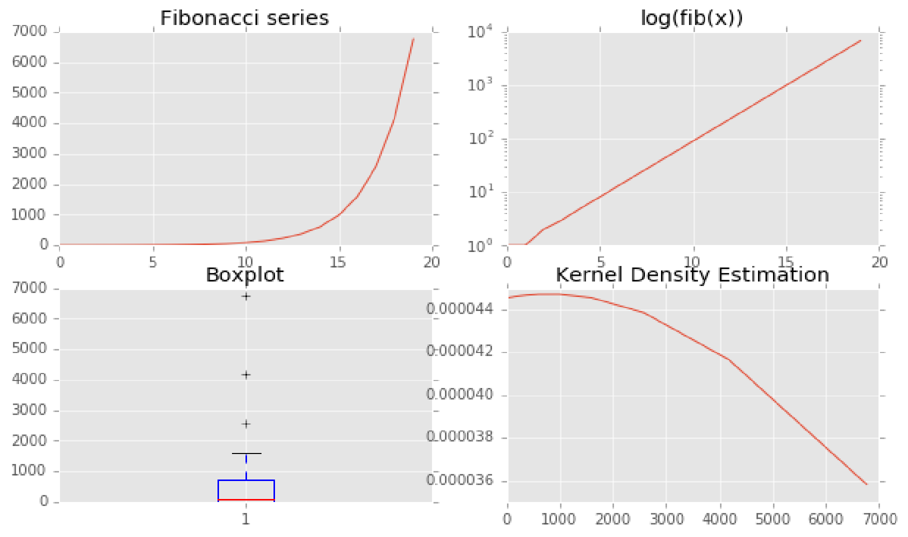
Matplotlib also features several colormaps which you can use:

```
from sklearn.datasets import make_blobs
X, y = make_blobs()

import matplotlib.pyplot as plt
import numpy as np
colors = plt.cm.rainbow(np.linspace(0, 1, 3))
plt.scatter(X[:, 0], X[:, 1], color=colors[y])
plt.show()
```



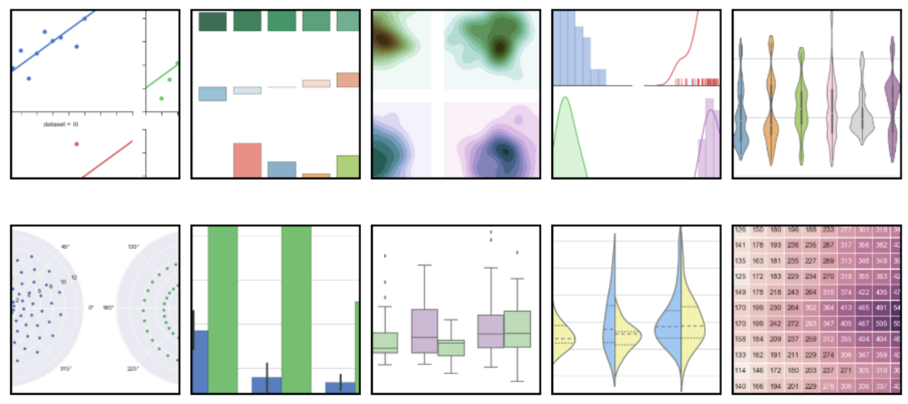
Matplotlib Exercise

Exercise: Plot your Fibonacci series like this:



Other Cool Packages for Data Visualization

seaborn

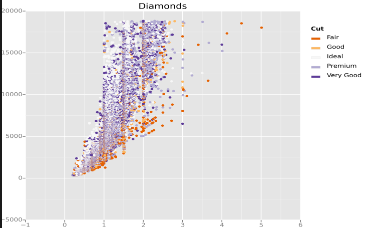


Seaborn extends matplotlib and contains a bunch of additional graphics, like `pairplot` and `corrplot`.

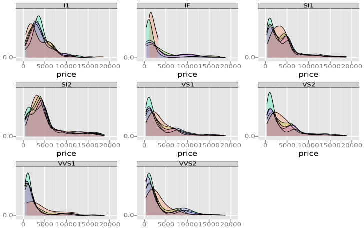
ggplot

For those who prefer the ggplot grammar, it is also available in python[5]. It works on top of matplotlib.

```
from ggplot import *
ggplot(diamonds,
       aes(x='carat', y='price',
       color='cut')) + \
    geom_point() + \
    scale_color_brewer(type='diverging',
                       palette=4) + \
    xlab('Carats') + ylab('Price') + \
    ggtitle('Diamonds')
```



```
from ggplot import *
ggplot(diamonds,
       aes(x='price', fill='cut')) + \
    geom_density(alpha=0.25) + \
    facet_wrap('clarity')
```

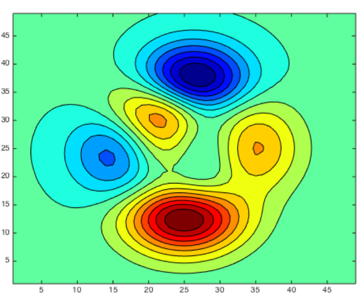


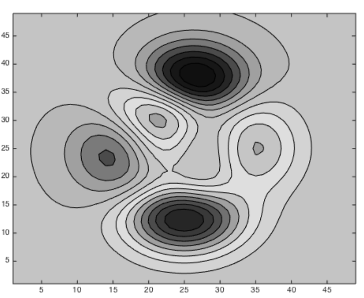
[5] http://ggplot.yhathq.com/

Other Notes on Data Visualization

Colormap

Previously, Matplotlib used a colormap called `jet`:



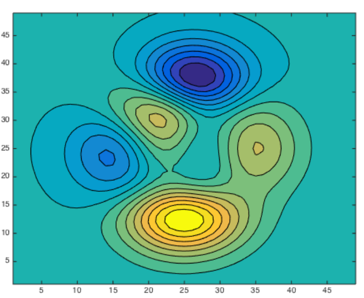


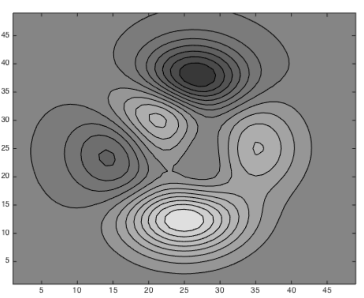
Problem:

- does not map to grayscale
- the scaling is not linear to the human eye
  - by changing slightly the data, the human eye can see things much differently

Colormap

MATLAB recently changed from `jet` to `parula`. Matplotlib will soon change from `jet` to `veridis`. (They are similar.)

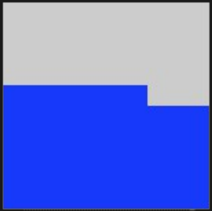




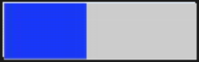
Graphics Choice

Some graphics are easier to the human eye.

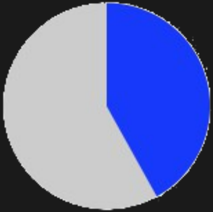
Which of the following graphics do you think makes it easier to communicate percentages?[6]



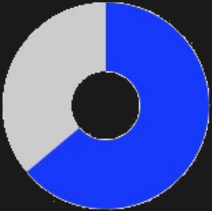
a) square chart



b) stacked bar



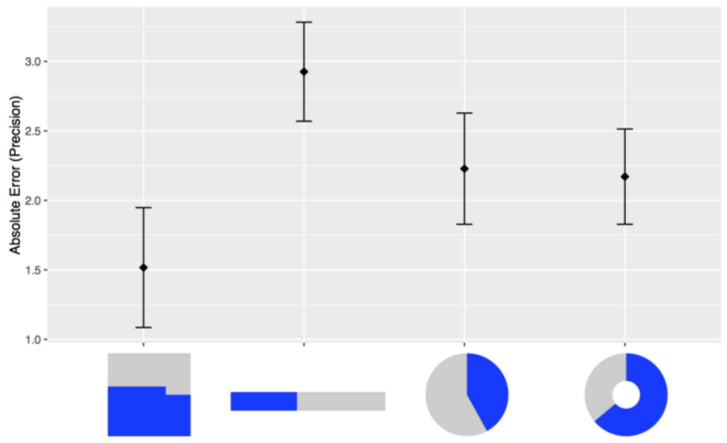
c) pie chart



d) donut chart

[6] http://blog.kaggle.com/2016/08/10/communicating-data-science-why-and-some-of-the-how-to-visualize-information/

Graphics Choice



Several studies have found stacked bar charts are awful to convey data.

All this to say that the choice of graphic is important to avoid lying with your data.

Summary

Python Limitations

All programming languages have limitations, right?

There are some computations that you can do in one language that you cannot do in another, and vice-versa.

Can you think of a program that you could write in MATLAB and not write in Python?

Actually, I fooled you.

Python, like any modern language, is Turing-complete.

What differentiates programming languages and their respective implementations is:

- how convenient the language is
- how efficient the implementation is

Numpy Exercise

6. Create a circle bitmask with sphere shape of radius r=4.

```
[[0 0 0 0 1 0 0 0 0]
 [0 0 1 1 1 1 1 0 0]
 [0 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 0]
 [1 1 1 1 1 1 1 1 1]
 [0 1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1 0]
 [0 0 1 1 1 1 1 0 0]
 [0 0 0 0 1 0 0 0 0]]
```

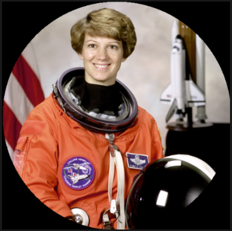
Suggestions:

```python
r = 4

bitmask = np.asarray([[
  np.uint8((x-r)**2+(y-r)**2 <= r**2)
    for y in range(0,r*2+1)]
      for x in range(0,r*2+1)])

# OR

x, y = np.meshgrid(range(0, r*2+1), range(0, r*2+1))
bitmask = ((x-r)**2 + (y-r)**2 <= r**2).astype(np.uint8)

# OR

y,x = np.ogrid[0:(r*2+1), 0:(r*2+1)]
bitmask = ((x-r)**2 + (y-r)**2 <= r**2).astype(np.uint8)
```

Numpy Exercise

7. Apply your circle bitmask to an image and show it (pixels outside of the circle should be black).

```
from skimage import data
img = data.astronaut()
w, h, _ = img.shape
y,x = np.ogrid[0:h, 0:w]
bitmask = (x-r-1)**2 + (y-r-1)**2 <= r**2
bitmask = bitmask.astype(np.uint8)
```



APIs: `np.repeat`, `np.tile`

Proceed from here...

Suggestions:

```
img2 = (img * np.repeat(bitmask, 3).reshape(w, h, 3))

img2 = img * bitmask[:, :, np.newaxis]

import matplotlib.pyplot as plt
plt.imshow(img2)
```

Homework

Please do the Numpy exercises available at:

`https://www.hackerrank.com/domains/python/py-introduction`