# Module 2, Day 2

Alireza Samar

UTM
UNIVERSITI TEKNOLOGI MALAYSIA

MLDS
MACHINE LEARNING
FOR DATA SCIENCE

# What We've Learned in Day 1

- Exploratory Data Analysis practices
- Pandas exercises

MLDS

# SciPy

Scientific Computing Toolkit

# What is SciPy

SciPy builds on top of NumPy to provide common tools for scientific programming, such as

- linear algebra
- numerical integration
- interpolation
- optimization
- distributions and random number generation
- signal processing
- etc., etc

MLDS

# What is SciPy

- Like NumPy, SciPy is stable, mature and widely used
- Many SciPy routines are thin wrappers around industry-standard Fortran libraries such as LAPACK, BLAS, etc.
- It's not really necessary to "learn" SciPy as a whole
- A more common approach is to get some idea of what's in the library and then look up documentation as required
- In this lecture we aim only to highlight some useful parts of the package

MLDS

# SciPy versus NumPy

- SciPy is a package that contains various tools that are built on top of NumPy, using its array data type and related functionality

MLDS

# SciPy versus NumPy

- In fact, when we import SciPy we also get NumPy, as can be seen from the SciPy initialization file

```
# Import numpy symbols to scipy name space
import numpy as _num
linalg = None
from numpy import *
from numpy.random import rand, randn
from numpy.fft import fft, ifft
from numpy.lib.scimath import *

__all__  = []
__all__ += _num.__all__
__all__ += ['randn', 'rand', 'fft', 'ifft']

del _num
# Remove the linalg imported from numpy so that the scipy.linalg package can be
# imported.
del linalg
__all__.remove('linalg')
```

MLDS

# SciPy versus NumPy

- However, it’s more common and better practice to use NumPy functionality explicitly

```
import numpy as np

a = np.identity(3)
```

MLDS

# SciPy versus NumPy

- What is useful in SciPy is the functionality in its subpackages
- *scipy.optimize, scipy.integrate, scipy.stats*, etc.
- These subpackages and their attributes need to be imported separately

```
from scipy.integrate import quad
from scipy.optimize import brentq
# etc
```

MLDS

Let’s explore some major packages.

MLDS

# Statistics

- Recall that ***numpy.random*** provides functions for generating random variables

```
In  import numpy as np

    np.random.beta(5, 5, size=3)
Out array([ 0.6167565 ,  0.67994589,  0.32346476])
```

- This generates a draw from the distribution below when *a, b = 5, 5*

$$f(x;a,b) = \frac{x^{(a-1)}(1-x)^{(b-1)}}{\int_0^1 u^{(a-1)}u^{(b-1)}du} \qquad (0 \le x \le 1)$$

MLDS

# Statistics

- Sometimes we need access to the density itself, or the cdf, the quantiles, etc.
- For this we can use ***scipy.stats***, which provides all of this functionality as well as random number generation in a single consistent interface

MLDS

# Statistics

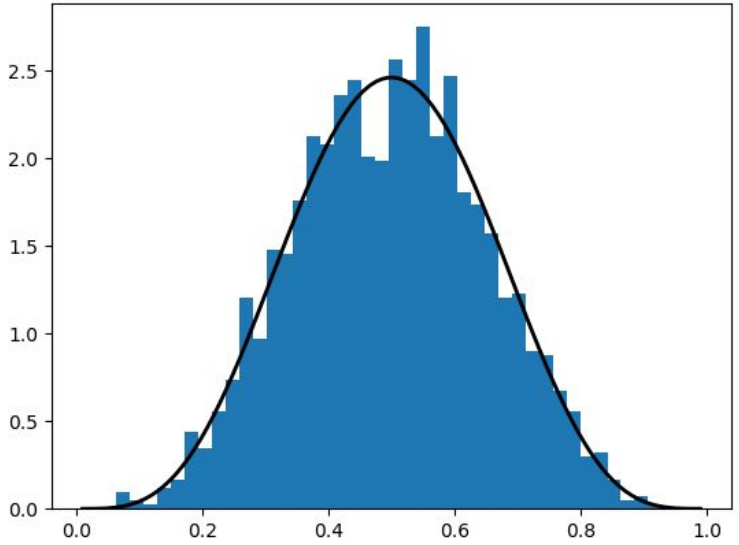
Here’s an example of usage

```
import numpy as np
from scipy.stats import beta
import matplotlib.pyplot as plt

q = beta(5, 5)      # Beta(a, b), with a = b = 5
obs = q.rvs(2000)   # 2000 observations
grid = np.linspace(0.01, 0.99, 100)

fig, ax = plt.subplots()
ax.hist(obs, bins=40, normed=True)
ax.plot(grid, q.pdf(grid), 'k-', linewidth=2)
plt.show()
```

MLDS

# Statistics

The following plot is produced




MLDS

# Statistics

- In this code we created a so-called ***rv_frozen*** object, via the call *q = beta(5, 5)*
- The “frozen” part of the notation implies that ***q*** represents a particular distribution with a particular set of parameters
- Once we’ve done so, we can then generate random numbers, evaluate the density, etc., all from this fixed distribution

MLDS

# Statistics

```
In  q.cdf(0.4)      # Cumulative distribution function
Out 0.2665676800000002

In  q.pdf(0.4)      # Density function
Out 2.0901888000000004

In  q.ppf(0.8)      # Quantile (inverse cdf) function
Out 0.63391348346427079

In  q.mean()
Out 0.5
```

MLDS

# Other Goodies in scipy.stats

- There are a variety statistical functions in ***scipy.stats***
- For example, ***scipy.stats.linregress*** implements simple linear regression

In
```
from scipy.stats import linregress

x = np.random.randn(200)
y = 2 * x + 0.1 * np.random.randn(200)
gradient, intercept, r_value, p_value, std_err = linregress(x, y)
gradient, intercept
```

Out
```
(1.9962554379482236, 0.008172822032671799)
```

MLDS

# Roots and Fixed Points

- A **root** of a real function $f$ on $[a,b]$ is an $x \in [a,b]$ such that $f(x)=0$

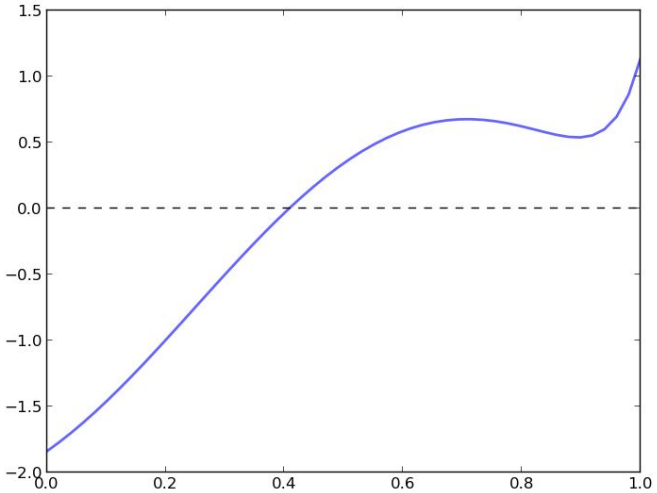
For example, if we plot the function

$$f(x)=\sin(4(x-1/4))+x+x^{20}-1$$

with $x \in [0,1]$ we get

MLDS

# Roots and Fixed Points




The unique root is approximately 0.408

Let’s consider some numerical techniques for finding roots

MLDS

# Bisection

One of the most common algorithms for numerical root finding is bisection

To understand the idea, recall the well known game where

- Player A thinks of a secret number between 1 and 100
- Player B asks if it's less than 50
    - If yes, B asks if it's less than 25
    - If no, B asks if it's less than 75

And so on

**This is bisection!**

MLDS

# Bisection

Here's a fairly simplistic implementation of the algorithm in Python

It works for all sufficiently well behaved increasing continuous functions with *f(a) < 0 < f(b)*

```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b

    while upper - lower > tol:
        middle = 0.5 * (upper + lower)
        # === if root is between lower and middle === #
        if f(middle) > 0:
            lower, upper = lower, middle
        # === if root is between middle and upper  === #
        else:
            lower, upper = middle, upper

    return 0.5 * (upper + lower)
```

MLDS

# Bisection

In fact SciPy provides it's own bisection function, which we now test using the function ***f*** defined before.

In
```
from scipy.optimize import bisect

f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1
bisect(f, 0, 1)
```

Out `0.40829350427936706`

MLDS

# The Newton-Raphson Method

Another very common root-finding algorithm is the *Newton-Raphson method*

In SciPy this algorithm is implemented by ***scipy.optimize.newton***

Unlike bisection, the Newton-Raphson method uses local slope information

```
In  from scipy.optimize import newton

    newton(f, 0.2)   # Start the search at initial condition x = 0.2
Out 0.40829350427935679

In  newton(f, 0.7)   # Start the search at x = 0.7 instead
Out 0.70017000000002816
```

MLDS

# The Newton-Raphson Method vs Bisection

- The second initial condition leads to failure of convergence
- On the other hand, using Jupyter's ***timeit*** magic, we see that newton can be much faster

```
In  %timeit bisect(f, 0, 1)
Out 1000 loops, best of 3: 261 us per loop

In  %timeit newton(f, 0.2)
Out 10000 loops, best of 3: 60.2 us per loop
```

MLDS

# Hybrid Methods

- So far we have seen that the Newton-Raphson method is fast but not robust
- This bisection algorithm is robust but relatively slow
- This illustrates a general principle
  - If you have specific knowledge about your function, you might be able to exploit it to generate efficiency
  - If not, then the algorithm choice involves a trade-off between speed of convergence and robustness
- In practice, most default algorithms for root finding, optimization and fixed points use hybrid methods
- These methods typically combine a fast method with a robust method in the following manner:
  - Attempt to use a fast method
  - Check diagnostics
  - If diagnostics are bad, then switch to a more robust algorithm

MLDS

# Hybrid Methods

- In *scipy.optimize*, the function ***brentq*** is such a hybrid method, and a good default

```
In  brentq(f, 0, 1)
Out 0.40829350427936706

In  %timeit brentq(f, 0, 1)
Out 10000 loops, best of 3: 63.2 us per loop
```

Here the correct solution is found and the speed is almost the same as *newton*

MLDS

# Fixed Points

SciPy has a function for finding (scalar) fixed points too

```
In  from scipy.optimize import fixed_point

    fixed_point(lambda x: x**2, 10.0)  # 10.0 is an initial guess
Out array(1.0)
```

If you don’t get good results, you can always switch back to the *brentq* root finder, since the fixed point of a function *f* is the root of $g(x) := x - f(x)f$

MLDS

# Optimization

- Most numerical packages provide only functions for *minimization*
- **Maximization** can be performed by recalling that the maximizer of a function $f$ on domain $D$ is the minimizer of $-f$ on $D$
- **Minimization** is closely related to root finding: For smooth functions, interior optima correspond to roots of the first derivative
- The speed/robustness trade-off described above is present with numerical optimization too
- Unless you have some prior information you can exploit, it's usually best to use hybrid methods

MLDS

# Optimization

- For constrained, univariate (i.e., scalar) minimization, a good hybrid option is ***fminbound***

```
In  from scipy.optimize import fminbound

    fminbound(lambda x: x**2, -1, 2)  # Search in [-1, 2]
Out 0.0
```

MLDS

# Integration

- Most numerical integration methods work by computing the integral of an approximating polynomial
- The resulting error depends on how well the polynomial fits the integrand, which in turn depends on how “regular” the integrand is
- In SciPy, the relevant module for numerical integration is ***scipy.integrate***

MLDS

# Integration

- A good default for univariate integration is ***quad***

```
In  from scipy.integrate import quad

    integral, error = quad(lambda x: x**2, 0, 1)
    integral
Out 0.33333333333333337
```

- In fact ***quad*** is an interface to a very standard numerical integration routine in the Fortran library QUADPACK
- There are other options for univariate integration—a useful one is ***fixed_quad***, which is fast and hence works well inside *for* loops

MLDS

# Flashback!

MLDS

# Recursive Function Calls

- This is not something that you will use every day, but it is still useful — you should learn it at some stage
- Basically, a recursive function is a function that calls itself
- For example, consider the problem of computing $x_t$ for some $t$ when

$$x_{t+1} = 2x_t, \quad x_0 = 1$$

Obviously the answer is $2^t$

MLDS

# Recursive Function Calls

We can compute this easily enough with a loop

In
```
def x_loop(t):
    x = 1
    for i in range(t):
        x = 2 * x
    return x
```

We can also use a recursive solution, as follows

In
```
def x(t):
    if t == 0:
        return 1
    else:
        return 2 * x(t-1)
```

MLDS

# Recursive Function Calls

- What happens here is that each successive call uses it’s own frame in the stack
  - a frame is where the local variables of a given function call are held
  - stack is memory used to process function calls
    - a First In Last Out (FILO) queue

MLDS

# Exercise!

MLDS

# Exercise

Write a recursive implementation of the bisection function described earlier

In

```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b

    while upper - lower > tol:
        middle = 0.5 * (upper + lower)
        # === if root is between lower and middle === #
        if f(middle) > 0:
            lower, upper = lower, middle
        # === if root is between middle and upper  === #
        else:
            lower, upper = middle, upper

    return 0.5 * (upper + lower)
```

MLDS

# Exercise

Test it on the function **f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1** discussed above

MLDS

# Exercise: Solution

Here's a reasonable solution:

In [1]:
```
def bisect(f, a, b, tol=10e-5):
    """
    Implements the bisection root finding algorithm, assuming that f is a
    real-valued function on [a, b] satisfying f(a) < 0 < f(b).
    """
    lower, upper = a, b
    if upper - lower < tol:
        return 0.5 * (upper + lower)
    else:
        middle = 0.5 * (upper + lower)
        print('Current mid point = {}'.format(middle))
        if f(middle) > 0:   # Implies root is between lower and middle
            bisect(f, lower, middle)
        else:               # Implies root is between middle and upper
            bisect(f, middle, upper)
```

MLDS

# Exercise: Solution

We can test it as follows

In [2]:
```
import numpy as np
f = lambda x: np.sin(4 * (x - 0.25)) + x + x**20 - 1

bisect(f, 0, 1)
```

```
Current mid point = 0.5
Current mid point = 0.25
Current mid point = 0.375
Current mid point = 0.4375
Current mid point = 0.40625
Current mid point = 0.421875
Current mid point = 0.4140625
Current mid point = 0.41015625
Current mid point = 0.408203125
Current mid point = 0.4091796875
Current mid point = 0.40869140625
Current mid point = 0.408447265625
Current mid point = 0.4083251953125
Current mid point = 0.40826416015625
```

MLDS

# Thanks!

Machine Learning for Data Science Interest Group
Advanced Informatics School
Universiti Teknologi Malaysia

@utmmlds
ais.utm.my/mlds

April 2017