

Introduction to Machine Learning (by Implementation)

Lecture 0: Python, Random Numbers, git

Ian J. Watson

University of Seoul

University of Seoul Graduate Course 2018



KRF KOREA RESEARCH FELLOWSHIP
해외 우수신진연구자 유치사업



- Welcome to "Introduction to Machine Learning (by Implementation)"
- Goals
 - Learn about some machine learning algorithms
 - Implement some machine learning algorithms
 - Start by using *no* ML libraries, just pure python
 - Thereby gain an understanding of the algorithms
 - The code we write will not be fast/stable/error-checking
 - Knowing the ideas, it should be easy to understand and adapt to your favourite library
 - If nothing else, have fun doing a lot of coding
- Grading:
 - Attendance
 - Completing projects each lesson
 - Final project

One slide overview of ML

- Goal of ML: get the computer to solve problems for us
 - I.e. we don't code an algorithm specifically for a problem, but write general algorithms by which the computer may "learn" from data
- Two main types of *supervised* problems, where you want the computer to generalize based on some *training* data, which has the answers or *labels*
 - Classification: given some data, does it belong to one category or another?
 - Classify images based on contents of image (this is a dog? a cat?)
 - Given some detector readings, what was the particle that impinged on the detector (electron? photon? muon?)
 - Regression: given some data, what was the underlying real variable that caused the data?
 - Given an image of a person, can you tell how old they are?
 - Given the raw calorimeter readings, what was the energy of the particle
- Also, have *unsupervised learning*, where the training data isn't labeled
 - Does the data tend to clump into categories? How many?
- We will be looking at these questions and seeing some answers people have discovered, but today ...

(Very Brief) Python Reminder

- I expect you've all seen python code before, but a whirlwind reminder:

```
# Comments begin with a '#' and go to the end of the line
```

```
# Variable assignment, calculator
```

```
a = 2 + 3
```

```
# Code blocks are indented, syntax starting blocks end line with a ':'
```

```
if a == 5: # Switch code paths with an if, 'truthy' values run code
    # in the 'if' block
```

```
    print("Yes it is!")
```

```
elif a == 4: # Can have multiple 'if'-like blocks run one after the
    # other (only the first one to be true is run)
```

```
    print("Almost")
```

```
else: # 'falsy' values run in (optional) else block. 'falsy' values
    # are: False, None, [], {}, "", set(), 0, 0.0. All other values
    # are true
```

```
    print("Nope nope nope")
```

```
# Functions defined with "def" block
def f(x, y, z):
    return x + y + z

# Called with usual syntax
print(f(1, 2, 4))

g = lambda x, y, z: x + y + z # Same as f above, syntax for simple fn's

def gcd(x, y):
    while x != y: # Use while to loop with a stopping condition
        if a > b:
            a = a - b
        else:
            b = b - a
    return a

def min(l):
    if len(l) == 0: raise TypeError # Raise errors on bad input
    mini = l[0]
    for x in l: # Use for to loop on lists
        if x < mini: mini = x
    return mini
```

List comprehensions

```
l = [1, 2, 3, 4]
```

```
# List comprehension example:
```

```
l_squared = [l_i*l_i for l_i in l]
```

```
# Same as the block:
```

```
output = []
```

```
for l_i in l:
```

```
    output.append(l_i*l_i)
```

```
l_squared = output
```

```
# Can add if conditions:
```

```
l_squared = [l_i*l_i for i, l_i in enumerate(l) if i % 2 == 0]
```

```
# Basically the same as:
```

```
output = []
```

```
for i, l_i in enumerate(l): # enumerate returns the index as well as  
                           # the value, i.e. i = 0, 1, 2, 3, etc
```

```
    if i % 2 == 0: # so we only take every second entry in the list
```

```
        output.append(l_i*l_i)
```

```
l_squared = output
```

An important organizational principle in python is the "class". Creates a new "datatype", from which you can create "objects". Objects keeps related data and methods (functions available to the object) together as a whole. The datatype defines what methods and data should exist.

```
class AClass: # class declares a new datatype, no objects of this
              # type are created yet
    def __init__(self): # methods can be attached to the class, first
                        # parameter is "self", the object itself
        self.a = 5 # kept in the object, can be used later
    def adder(self, n):
        return self.a + n
```

```
# Create a new object of type "AClass"
an_obj = AClass() # calls the __init__ method on creation
an_obj.a # 5
an_obj.adder(7) # 12, "self" passed automatically (here an_obj)
```

Python2 vs Python3

- Main differences:
 - Python 2 support from major libraries (eg numpy) being stopped!
 - Python 3 uses `print` **function**, i.e. need the parens!
 - Python 3 uses float division with `/`, integer division with `//` **always**
 - Python 2 `/` would do int or float depending on the args
 - Python 3 strings are quite different
 - Unless you're using strings as byte arrays, or doing unicode work, shouldn't notice
 - If you are and want to understand all the `.encode()` / `.decode()`, let me know

Type Signatures

- Python has a utility `mypy` which can check types (similar to what C++ does)
- In python3, can use the syntax:

```
def aFn(a: input1_type, b: input2_type) -> output_type:  
    c: int_type = fn(a, b)
```

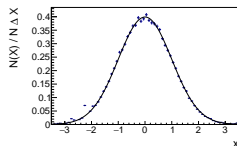
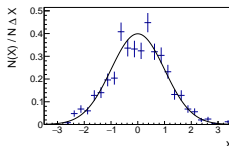
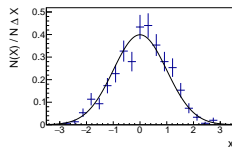
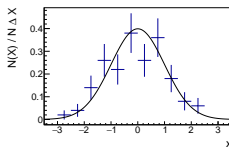
- where `input1_type`, `input2_type`, etc. are python objects
- By itself, doesn't do anything, just user-defined meta-data attached to objects
- However, `mypy` interprets them as types, and checks that the types match up
 - If no type is present, it assumes that it is of type `Any`, which represents any type
- Can use `List[atype]` to represent a list of `atype` objects
 - `from typing import List`
- I will use this syntax to tell you what the types the functions you write should include
 - You can add them or not, but, beware, in python2, the special syntax doesn't work

What do we mean by randomness?

- In ML, we often need random numbers when developing models
 - Starting positions for parameter searches, initial model parameters, etc.
- The basic idea is that we want a *random number generator*
 - In practice, a black-box function we can call that returns numbers
- There shouldn't be a way to predict numbers from the generator better than chance
- We also want random number generators for different PDFs
 - PDF: Probability Distribution Function, see my last course "Practical Statistics for Particle Physicists"
 - The random numbers should be distributed according to the PDF
 - E.g. Neural networks perform well with seeds from a gaussian PDF

Distributions of random numbers

- What do we mean by "distributed according to a PDF"?



- After one random number, we can't tell much of anything about it:
 - Obligatory xkcd:
 - ```
int getRandomNumber() { return 4; /* chosen by a fair dice roll, guaranteed random */ }
```
- As we take random numbers from our generator, the (normalized) distribution of these number should approach our ideal PDF
  - At infinity, should be indistinguishable from the PDF
- In practice, we don't have time to generate an infinite amount of random numbers to test, so we have statistical tests of randomness

# Sources of randomness

- Where do we get random numbers?
- Could toss a coin (single bit of randomness per coin toss), or roll a dice
  - Doesn't really scale to millions of numbers
- Could attach a quantum device to the computer
  - Prepare a Schroedingers cat type state, check if the cat's alive or dead
  - E.g. for genuine random number on demand based on radioactive decay: <https://www.fourmilab.ch/hotbits/>
- Similarly, could use chaotic systems (thermal noise, atmospheric noise)
- Such hardware devices, True Random Number Generator (TRNG), do exist or can be implemented through clever repurposing, but tend to be slow or expensive (pick one)
- But, we need a way to create millions of random numbers a second

# Pseudo-random numbers

- In practice, we don't need "truly" random numbers, just number sequences with the right properties
  - No correlations in the random numbers produced, non-repeating, for any given number, same prob. to get it as any other number
- Thus were "pseudo-random number generators" produced
- The idea is to start with some seed data (taken from wherever), then pass that through some function to produce a "random" number and a new state to seed the next number (possibly just the number itself)
- With a carefully chosen function, the output sequence has the properties we desire
- These are "Pseudo-Random Number Generators" (PRNG)

# Example: Linear Congruential Generator

- One of the oldest and best-known algorithms
- Start with a seed number  $X_0$ , then generate new numbers by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

- $a$ ,  $c$ , and  $m$  are constants which must be judiciously chosen to avoid repeating sequences
- Advantages: fast, only need to keep last number generated
- Disadvantages: periodic (if you hit a number you've seen before, the sequence replays exactly the same), poor choices of the constants lead to bad performance
  - If a number already produced appears again, the sequence starts over
    - Will happen on average after  $\sqrt{m}$  numbers, by the birthday paradox
  - Many early RNG libraries had bad choices, leading to statistical errors in papers!

# Example: RANDU

- A widely distributed algorithm in wide use since the 1960s
  - In use until the late 90s, these days, newer methods such as Mersenne Prime Twisters are used (TRandom3 in ROOT)
- Uses LCG to generate floating point numbers in  $[0, 1)$ 
  - Floating point is a whole other issue

$$V_{j+1} = 65539 \cdot V_j \mod 2^{31}$$

$$X_j = V_j / 2^{31}$$

- The initial seed should be an odd number
- This had a multi-dimensional correlation that led to incorrect results (try generating 3-dim. points and plotting in 3D, should see that it generates in planes)
  - Don't trust results from the 80s and even into the 90s that use the distribution "standard" PRNG!

# Exercises

In Machine Learning, we often need random numbers to seed our models, today, we will write some:

- ❶ Implement a linear congruential generator in python, with input integer and output float in range  $[0, 1]$ 
  - The book "Numerical Recipes" suggests using LCG with  $a = 1664525$ ,  $c = 1013904223$ ,  $m = 2^{32}$ , use these values
  - Create a class `randnr`, it should have an initializer that sets the seed
    - `def __init__(self, seed: int):`
  - With a function to get a random integer in range  $(0, 2^{32})$ :
    - `def randint(self) -> int:`
  - And a random number output function for floats in range  $(0, 1)$ :
    - `def random(self) -> float:`
- ❷ Using your `randnr` function, implement an exponential PDF
  - If you take output from `randnr`, `r` and feed into `-c*math.log(1-r)`, it outputs a random number from the PDF  $f(x) = \frac{1}{c}e^{-x/c}$
  - add to your class the function:
    - `def exp(self) -> float:`



## 1 Using your RANDNR function, implement a Gaussian PDF Generator

- The Central Limit Theorem of probability tells us that the sum of random distributions approaches a Gaussian (normal) distribution
- In fact, if  $x_i$  is a random variable uniform in  $(0, 1)$ , then

$$x = \frac{\sum_{i=1}^m x_i - m/2}{\sqrt{m/12}} \rightarrow \text{Gaussian as } m \rightarrow \infty$$

(where does  $\sqrt{1/12}$  come from?)

- Where the gaussian has mean 0, standard deviation 1,  $N(0, 1)$
- So, add the function `def gauss(self, mean=0, std=1, m=10) -> float:`, which takes `m` samples from `randnr.random` and runs the gaussian approximation sum, then scales the output for the given mean and standard deviation  $N(\mu, \sigma) = \sigma \cdot N(0, 1) + \mu$
- [Optional for students without prior experience] Draw a histogram (matplotlib/ROOT) of the output of `gauss` for `m=1, 2, 5, 10`, filling each histogram with 1000 values pulled from the distribution, plot the on the same axes and save to a file called 'gauss.png', include it in your github repo

# Submit your code:

- Code will be marked automatically using github classrooms
  - You will create a git repository which I will use to mark automatically
  - It contains a test file that checks your code works as it should
    - The class `randnr` exists, and that `randint` and `random` deliver numbers from the NR LCG, and the `exp` is working correctly
  - If you pass all the tests, you are done for today
  - Run the tests by running `pytest`
    - If it isn't installed, run `pip install pytest --user`
- Create a github account
- <https://classroom.github.com/a/vKNBpsBj>
- Click the first assignment link to create your repo
- Clone the repo (instructions on the page)
- Write code, test, upload to repo
  - I will mark tomorrow based on the last commit with a timestamp of today
  - Bonus marks: add tests to the `test_randomness.py` file to check that your code is working properly. I will merge the tests together when making the final version for code marking