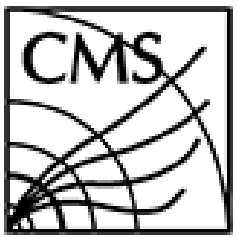
# Introduction to Machine Learning (by Implementation)

## Lecture 1: Vectors and Matrices

Ian J. Watson

University of Seoul

University of Seoul Graduate Course 2018




# Programming on the Command Line

- Today, we will focus on just churning out code, the project will be to implement basic vector and matrix operations
- Get everyone use to writing classes and function
- To begin, I'll try to clear up a few issues encountered last week
- Let's start with some pointers when writing programs/libraries rather than running python through Jupyter:

```
if __name__ == "__main__":
    print("This code will only run if we run the file on the command line")
    print("If we import the file (in the interpreter or another file)")
    print("It won't run.")
    print("In that case __name__ will be the name of the import'ed module")
    print("Use this when you want to test/make code sometimes, ")
    print("but will usually use the file it as a library")
```

# Import (very brief overview)

- When writing lots of code, it can be useful to write in separate files
- From one file, you can `import` another file (don't add the .py extension)
- `import` looks for python files in the current directory, the system libraries, and directories in your `PYTHONPATH` env. var.
- Say you have two files in your directory:

Library.py

```
a = 2
def function():
  print("Very important function")
if __name__ == "__main__":
  b = 4
  print("b won't be seen if you import Library")
```

Runner.py

```
import Library  # Will find Library.py in the same directory
Library.function()  # You can run your important functions
RunnerA = Library.a + 5  # This works
RunnerB = Library.b + 7  # ERROR: b isn't defined when imported
```

# class

```
# Every class *derives from* (can use the features of) object
class AC(object):  # if you want features of another class, change "object"
  # when A() is called a fresh object is made, __init__ code runs on it
  def __init__(self): self.a =5; self.b = 6
  # If you call the function from "object.", self is passed implicitly
  def aFunction(self, b): return self.a + self.b + b

aobj = AC()  # Creates the object, calls the A.__init__ method
aobj.a  # 5, aobj.a accesses the "member variable" a of aobj
aobj.aFunction(2)  # 5 + 6 + 2 = 13, "method" of aobj
AC.aFunction(aobj, 2)  # equivalent to the above line
```

- class "members" may be attached and updated through the 'dot' operator
  - Each class object has its own set of member variables, separate from other objects of the same class
- class "methods" (functions) live inside the class (`AC`) and the object (`aobj`), where the object version implicitly sets `self`

# virtualenv

- `pip install --user virtualenv`
- Then, can be run from `~/.local/bin/vritualenv` (usually)
- `virtualenv ENV` will create a "virtual environment" of python in `ENV`
- `source ENV/bin/activate` sets up your command line to use it
- Python exe and packages preferably taken from ENV, rather than system
  - `pip install <package>` installs package to the virtualenv ENV instead of globally
  - Effectively ENV creates an isolated python separate from the system
  - Can avoid issues of bad library paths
- If your `pytest` command has issues, `pip` doesn't work, try running a virtualenv
- (NB CMSSW setups its own python, ROOT needs very specific versions, a bit hard in particle physics...)

# Linear Algebra

- Today, we'll do some more coding warm-up, but in an area of importance to machine learning: Linear Algebra
- Linear Algebra is, of course, the study of linear systems, and their representation as *vector spaces*
- *Vectors* are a set equipped with a concept of addition, and scalar multiple, $(V, +, \cdot)$
  - Addition: If $a, b \in V$, then $a + b \in V$
  - Scalar multiple: If $a \in V$ and $c \in R$, then $c \cdot a \in V$
  - As well as various associative and inverse laws
- Objects could be vectors, matrices etc.
- Today, we'll implement some basic functions on vectors and matrices
- Give us some more coding exercises, but also, being able to write such functions will be very important in machine learning
  - E.g. Neural Networks are basically just stacks of linear operators

# Vectors

- Vectors (for us) are basically a list of numbers, $\mathbf{a} = (a_0, a_1 \ldots a_n)$
  - Formally, linear combinations of some basis vectors $\mathbf{a} = \sum_i a_i \mathbf{e_i}$
  - Could represent measurements we want to use to distinguish categories
    - For example, Length and width of petals to classify types of irises
- So, e.g. an object in $R^3$ could be represented as $(x, y, z)$
- We will represent a vector as a python list, matrix as a list of lists
  - Entries in the list should be floats
  - `Vector = List[float]` (using the syntax for types from last week)
  - `Matrix = List[List[float]]`
- These functions will of course, be slow
  - Point is just to write them in anger as an exercise
- Usually, you would use something built on top of BLAS, or so, which provide highly-optimized linear algebra routines
  - E.g. `numpy` is the standard python package which has a nice interface and is fast. Underneath it uses these libraries
- Or, if you're deep learning, something which compiles to CUDA, which can be run on a GPU for extreme performance

# Some basic functions to implement:

- start in the file linalg.py, and create the class `Vector`, which has:
- `__init__(self, initial: [float])`: takes in the list of numbers for the vector and stores them in `self`
- `shape(self: Vector) -> int`
  - the dimension of the vector (i.e. length of the list, in numpy terms, `shape` refers to general tensor size, here return a number)
- `i(self: Vector, i: int)` returns the (0-indexed) ith element
- `add(self: Vector, w: Vector) -> Vector`
  - each entry of the output (an new `Vector`) should $v_i + w_i$
- `subtract(self: Vector, w: Vector) -> Vector`
  - each entry of the output should be the subtraction of the corresponding entries of the input, $v_i - w_i$
- `scalar_multiply(self: Vector, c: float) -> Vector`
  - each entry of the output should be c multiplied by the input, $c \cdot v_i$
- using the interface above (i.e. only using those function), the function `mean(self: Vector) -> float`
  - Returns the mean of the entries in the vector

# More operations on vectors

- `dot(self: Vector, w: Vector) -> float` the dot product of the two vectors (sum of entries multiplied together)
- using `dot` and `math.sqrt` write `norm(self: Vector) -> float`, which returns the Euclidean norm (or magnitude, or length) of the vector
  - You have to `import math` (or write a square root function yourself)
- using the vector function (i.e. don't manipulate python lists), write `distance(self, w)` which returns the distance between the two vectors (or norm of the vector from v to w if you prefer)
- write the function (outside of the Vector class) `vector_sum(l: List[Vector])`, which sums the list of `Vector`. Use only the vector functions, not the representation as a list
- One point: notice that having and using `shape()` and `i(self, i)` allow us to abstract the underlying representation of the Vector
  - We could change the List for numpy array, and not have to rewrite code which uses `Vector`
  - This is the idea *encapsulation*, hiding the implementation behind an an API (Application Programming Interface)

# Matrices

- Matrices (for our purposes) are basically 2D tables
  - `Matrix = List[List[float]]`
- Could represent e.g. correlation coefficients between our variables
- We can add, subtract, scalar and vector and matrix multiply them
- Start the class `Matrix`, which is initialized with a list of lists of floats
  - eg `m = Matrix([ [1, 2], [3, 4] ])` for $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$
- `shape(self: Matrix) -> (int, int):` returns the size of the matrix (two-tuple)
- `ij(self: Matrix, i: int, j: int)`, 0-indexed element
- `identity(n: int) -> Matrix` return the $n \times n$ identity matrix (1 on the diagonal, 0 elsewhere), outside of the `Matrix` class
- `scalar_multiply(self: Matrix, c: float) -> Matrix`
- `vector_multiply(self: Matrix, v: Vector) -> Vector`
  - Calculates output vector o by $o_i = \sum_j M_{ij} v_j$
  - For now, we'll ignore distinction between row and column vector

# Matrices (continued)

- Write `transpose(self: Matrix) -> Matrix`, matrix transpose, the output $O$ is $O_{ij} = M_{ji}$
- `multiply(self: Matrix, B: Matrix) -> Matrix`
  - matrix multiplication, $O_{ij} = \sum_k A_{ik} B_{kj}$ where A = `self` here
- Some hints:
  - Check the test_linalg.py file if you get stuck, it should help you to see examples in use if something is unclear (there are lots of examples)
  - There will be a lot of manipulating of lists, to begin, try writing list comprehensions, but if you get confused (particularly for Matrix), explicitly write out all the loops
    - What do you need to create the output matrix? Can you build it up step by step?
  - In python 3, list comprehensions **don't** return lists, it returns a generator, to make the actual list, need to call the `list` function
    - `[a for a in [1, 2, 3]]` is a generator (tells how to make the list)
    - `list([a for a in [1, 2, 3]])` makes the actual list
    - `list(a for a in [1, 2, 3])` same as above, don't need the square brackets
- For Ian: show setup venv and write `class Vector` up to `i()`