CS 5489 Machine Learning

Lecture 1b: Numpy, Matplotlib

Prof. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. Control structures (conditional and loops)
4. Functions, Classes
5. File IO, Pickle, pandas
6. **NumPy**
7. matplotlib
8. probability review

NumPy

- Library for multidimensional arrays and 2D matrices
- `ndarray` class for multidimensional arrays
  - elements are all the same type
  - aliased to `array`

In [1]:
```
from numpy import *      # import all classes from numpy
a = arange(15)
a
```

Out[1]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

In [2]:
```
b = a.reshape(3,5)  # rows x columns
b
```

Out[2]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [3]:
```
b.shape  # get the shape (num rows x num columns)
```

Out[3]:
```
(3, 5)
```

In [4]:
```
b.ndim   # get number of dimensions
```

Out[4]:
```
2
```

In [5]:
```
b.size   # get number of elements
```

Out[5]:
```
15
```

In [6]:
```
b.dtype  # get the element type
```

Out[6]:
```
dtype('int64')
```

Array Creation

In [7]:
```
a = array([1, 2, 3, 4])        # use a list to initialize
a
```

Out[7]:
```
array([1, 2, 3, 4])
```

In [8]:
```
b = array([[1.1,2,3], [4,5,6]]) # or list of lists
b
```

Out[8]:
```
array([[1.1, 2. , 3. ],
       [4. , 5. , 6. ]])
```

In [9]:
```
zeros( (3,4) )    # 3x4 array of zeros
```

Out[9]:
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

In [10]:
```
ones( (2,4) )   # 2x4 array of ones
```

Out[10]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [11]:
```
full( (3,4), 8.8)  # 3x4 array with all 8.8
```

Out[11]:
```
array([[8.8, 8.8, 8.8, 8.8],
       [8.8, 8.8, 8.8, 8.8],
       [8.8, 8.8, 8.8, 8.8]])
```

In [12]:
```
empty( (2,3) )  # create an array, but do not prepopulate it.
                # contents are random
```

Out[12]:
```
array([[1.1, 2. , 3. ],
       [4. , 5. , 6. ]])
```

In [13]:
```
arange(0,5,0.5)    # from 0 to 5 (exclusive), increment by 0.5
```

Out[13]:
```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5])
```

In [14]:
```
linspace(0,1,10)  # 10 evenly-spaced numbers between 0 to 1 (inclusive)
```

Out[14]:
```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

In [15]:
```
logspace(-3,3,13)  # 13 numbers evenly spaced in log-space between 1e-3 and 1e3
```

Out[15]:
```
array([1.00000000e-03, 3.16227766e-03, 1.00000000e-02, 3.16227766e-02,
       1.00000000e-01, 3.16227766e-01, 1.00000000e+00, 3.16227766e+00,
       1.00000000e+01, 3.16227766e+01, 1.00000000e+02, 3.16227766e+02,
       1.00000000e+03])
```

Array Indexing

- One-dimensional arrays are indexed, sliced, and iterated similar to Python lists.

In [16]:
```
a = array([1,2,3,4,5])
a[2]
```

Out[16]:
```
3
```

In [17]:
```
a[2:5]            # index 2 through 4
```

Out[17]: array([3, 4, 5])

In [18]:
```python
a[0:5:2]            # index 0 through 4, by 2
```

Out[18]: array([1, 3, 5])

In [19]:
```python
# iterating with loop
for i in a:
    print(i)
```

```
1
2
3
4
5
```

- For multi-dimensional arrays, each axis had an index.
  - indices are given using tuples (separated by commas)

In [20]:
```python
a = array([[1, 2, 3], [4, 5, 6], [7,8,9]])
print(a)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

In [21]:
```python
a[0,1]     # row 0, column 1
```

Out[21]: 2

In [22]:
```python
a[:,1]     # all elements in column 1
```

Out[22]: array([2, 5, 8])

In [23]:
```python
a[0:2, 1:3]   # sub array: rows 0-1, and columns 1-2
```

Out[23]:
```
array([[2, 3],
       [5, 6]])
```

In [24]:
```python
# "for" iterates over the first index (rows)
for r in a:
    print("--")
    print(r)
```

```
--
[1 2 3]
--
[4 5 6]
--
[7 8 9]
```

- indexing with a boolean mask

In [25]:
```python
a = array([3, 1, 2, 4])
m = array([True, False, False, True])
print("m =", m)
a[m]            # select with a mask
```

```
m = [ True False False  True]
```

Out[25]: array([3, 4])

multi-dimensional arrays (tensors)

- 3 x 2 x 4 tensor

- prints as three 2x4 arrays
- last index is iterated first

In [26]:
```
a = arange(24)
b = a.reshape((3,2,4))
print(b)
```

```
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]
```

- indexing is similar to 2-dim arrays (i,j,k)

In [27]:
```
b[2,0,1]
```

Out[27]:
```
17
```

- extract a "slice"

In [28]:
```
b[1,:]   # i=1
```

Out[28]:
```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [29]:
```
b[:,1,:]   # j=1
```

Out[29]:
```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15],
       [20, 21, 22, 23]])
```

In [30]:
```
b[:,:,1]   # k=1
```

Out[30]:
```
array([[ 1,  5],
       [ 9, 13],
       [17, 21]])
```

In [31]:
```
# iterate over the first index
for s in b:
    print("--")
    print(s)
```

```
--
[[0 1 2 3]
 [4 5 6 7]]
--
[[ 8  9 10 11]
 [12 13 14 15]]
--
[[16 17 18 19]
 [20 21 22 23]]
```

Array Shape Manipulation

- The shape of an array can be changed

In [32]:
```
a = array([[1,2,3], [4, 5, 6]])
print(a)
a.shape
```

```
[[1 2 3]
 [4 5 6]]
```

Out[32]: (2, 3)

In [33]:
```
a.ravel()      # return flattened array (last index iterated first).
```

Out[33]:
```
array([1, 2, 3, 4, 5, 6])
```

In [34]:
```
a.transpose()  # return transposed array (swap rows and columns)
```

Out[34]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [35]:
```
a.reshape(3,2)  # return reshaped array
```

Out[35]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

In [36]:
```
a.resize(3,2)   # change the shape directly (modifies a)
print(a)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

Concatenating arrays

In [37]:
```
a = array([1, 2, 3])
b = array([4, 5, 6])
concatenate((a,b))
```

Out[37]:
```
array([1, 2, 3, 4, 5, 6])
```

In [38]:
```
c_[a,b]       # concatenate as column vectors
```

Out[38]:
```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

In [39]:
```
r_[a,b]       # concatenate as row vectors
```

Out[39]:
```
array([1, 2, 3, 4, 5, 6])
```

Stacking arrays

In [40]:
```
a = array([[1, 1],
           [1, 1]])
b = array([[2, 2],
           [2, 2]])
vstack( (a,b) )      # stack vertically
```

Out[40]:
```
array([[1, 1],
       [1, 1],
       [2, 2],
       [2, 2]])
```

In [41]:
```
hstack( (a,b) )      # stack horizontally
```

Out[41]:
```
array([[1, 1, 2, 2],
       [1, 1, 2, 2]])
```

Array Operations

operators are applied **elementwise**

In [42]:
```
a = array( [20,30,40,50] )
b = arange( 4 )    # [0 1 2 3]
a - b              # element-wise subtraction
```

Out[42]:
```
array([20, 29, 38, 47])
```

In [43]:
```
b**2               # element-wise exponentiation
```

Out[43]:
```
array([0, 1, 4, 9])
```

In [44]:
```
10*sin(a)          # element-wise product and sin
```

Out[44]:
```
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

In [45]:
```
a < 35             # element-wise comparison
```

Out[45]:
```
array([ True,  True, False, False])
```

- product operator (`*`) is **elementwise**
  - i.e., Hadamard product

In [46]:
```
A = array( [[1,1],
            [0,1]] )
B = array( [[2,0],
            [3,4]] )
A*B                          # elementwise product
```

Out[46]:
```
array([[2, 0],
       [0, 4]])
```

- compound assignment: `*=`, `+=`, `-=`
- unary operators

In [47]:
```
a = array( [[1,2,3], [4, 5, 6]])
a.sum()
```

Out[47]:
```
21
```

In [48]:
```
a.min()
```

Out[48]:
```
1
```

In [49]:
```
a.max()
```

Out[49]:
```
6
```

- unary operators on each axis of array

In [50]:
```
a = array( [[1,2,3], [4, 5, 6]])
a.sum(axis=0)     # sum over rows
```

Out[50]:
```
array([5, 7, 9])
```

In [51]:
```
a.sum(axis=1)     # sum over column
```

Out[51]:
```
array([ 6, 15])
```

- Numpy provides functions for other operations (called universal functions)
  - `argmax`, `argmin`, `min`, `max`
  - `average`, `cov`, `std`, `mean`, `median`,
  - `ceil`, `floor`
  - `cumsum`, `cumprod`, `diff`, `sum`, `prod`

- `inv`, `dot`, `trace`, `transpose`

Broadcasting

- any binary operators (+, -, *, etc)
- if the two operands are not the same size
  - broadcasting tries to extend the singleton dimensions of one operand to match the other operand.
  - an Error is thrown if two operands can't be broadcast together.
- operands do not need to have the same number of dimensions
  - match dimensions from the right

In [52]:
```
a = array( [[1,2,3],
            [4,5,6]] )
```

In [53]:
```
b = array( [1,2,3] )
```

- a and b are not the same dimensions,
  - b is "stretched" so that it fills in a 2x3 shape

```
a:      2 x 3
b:          3
result: 2 x 3
```

In [54]:
```
a + b
```

Out[54]:
```
array([[2, 4, 6],
       [5, 7, 9]])
```

- c is stretched so that it fills in a 2x3 shape

```
a:      2 x 3
c:      2 x 1
result: 2 x 3
```

In [55]:
```
c = array( [[1],
            [2]] )
```

In [56]:
```
a+c
```

Out[56]:
```
array([[2, 3, 4],
       [6, 7, 8]])
```

- b and c are both stretched to 2x3 shape

```
b:          3
c:      2 x 1
result: 2 x 3
```

In [57]:
```
b+c
```

Out[57]:
```
array([[2, 3, 4],
       [3, 4, 5]])
```

- "newaxis" can insert an extra dimension

```
b:               3
b[:,newaxis]: 3 x 1
result:       3 x 3
```

In [58]:
```
b + b[:,newaxis]
```

Out[58]:
```
array([[2, 3, 4],
       [3, 4, 5],
```

```
       [4, 5, 6]])
```

Brief Linear Algebra Review

- column vector:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} \in \mathbb{R}^d$$

- matrix:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,n} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

- matrix as collection of column vectors: $\mathbf{A} = \begin{bmatrix} | & & | \\ \mathbf{a}_1 & \cdots & \mathbf{a}_n \\ | & & | \end{bmatrix}$
  - $\mathbf{a}_i$ is the i-th column of $\mathbf{A}$.

In [59]:
```python
x = array([1,2,3]).reshape((3,1))
print(x)
```

```
[[1]
 [2]
 [3]]
```

In [60]:
```python
A = zeros((3,3))
print(A)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

- Transpose: swap rows and columns
  - $\mathbf{x}^T = [x_1 \cdots x_d]$

In [61]:
```python
z = x.transpose()
print(z)
```

```
[[1 2 3]]
```

Inner product

- Inner product: $\mathbf{x}^T\mathbf{y} = \sum_{i=1}^d x_i y_i$
  - measures the similarity between vectors $\mathbf{x}$ and $\mathbf{y}$.

In [62]:
```python
x = array([1, 2, 3])
y = array([2, 1, 1])
inner(x,y)
```

Out[62]: 7

- Length (norm):

$$||\mathbf{x}|| = \sqrt{\mathbf{x}^T\mathbf{x}} = \sqrt{\sum_{i=1}^d x_i^2}$$

In [63]:
```
x = array([1, 2, 3])
linalg.norm(x)
```
Out[63]:
```
3.7416573867739413
```

- Distance between two vectors:

$$||\mathbf{x}-\mathbf{y}|| = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$$

In [64]:
```
y = array([2, 1, 1])
linalg.norm(x-y)
```
Out[64]:
```
2.449489742783178
```

- Outerproduct between two vectors: $\mathbf{x}\mathbf{y}^T = \begin{bmatrix} y_1\mathbf{x} & \cdots & y_d\mathbf{x} \end{bmatrix}$

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} x_1y_1 & \cdots & x_1y_d \\ \vdots & \ddots & \vdots \\ x_dy_1 & \cdots & x_dy_d \end{bmatrix}$$

In [65]:
```
x = array([1, 2, 3])
y = array([2, 1, 1])
outer(x,y)
```
Out[65]:
```
array([[2, 1, 1],
       [4, 2, 2],
       [6, 3, 3]])
```

Matrix multiplication

- need compatible dimensions: $\mathbf{C}_{m\times n} = \mathbf{A}_{m\times d}\mathbf{B}_{d\times n}$

$$\mathbf{A} = \begin{bmatrix} \boxed{a_{1,1} \quad \cdots \quad a_{1,n}} \\ \vdots \quad \ddots \quad \vdots \\ a_{m,1} \quad \cdots \quad a_{m,n} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \boxed{\begin{matrix} b_{1,1} \\ \vdots \\ b_{m,1} \end{matrix}} & \begin{matrix} \cdots \\ \ddots \\ \cdots \end{matrix} & \begin{matrix} b_{1,n} \\ \vdots \\ b_{m,n} \end{matrix} \end{bmatrix}$$

- Entry in $\mathbf{C}$:

$$c_{i,j} = \mathbf{a}_i\mathbf{b}_j = \sum_{k=1}^{d} a_{i,d}b_{d,j}$$

In [66]:
```
A = array([[1, 2, 3],
           [2, 1, 0]])
B = array([[-1, 1],
           [0,  1],
           [1,  0]])
A @ B
```
Out[66]:
```
array([[ 2,  3],
       [-2,  3]])
```

Matrix-Vector multiplication

- Different interpretations if using transpose or not.

- $\mathbf{Ax}$: Linear combination of the columns of $\mathbf{A}$
  - $\mathbf{A} \in \mathbb{R}^{m\times d}, \mathbf{x} \in \mathbb{R}^d$:

$$\mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} | & & | \\ \mathbf{a}_1 & \cdots & \mathbf{a}_d \\ | & & | \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \sum_{i=1}^{d} x_i \mathbf{a}_i \in \mathbb{R}^m$$

In [67]:
```
A = array([[1, 2],
           [3, 5]])
x = array([-1, 1])
A @ x  # matrix multiplicattion
```

Out[67]:
```
array([1, 2])
```

- $\mathbf{A}^T\mathbf{x}$: Vector of inner products with columns of $\mathbf{A}$
  - $\mathbf{A} \in \mathbb{R}^{d\times m}, \mathbf{x} \in \mathbb{R}^d$:

$$\mathbf{y} = \mathbf{A}^T\mathbf{x} = \begin{bmatrix} | & & | \\ \mathbf{a}_1 & \cdots & \mathbf{a}_d \\ | & & | \end{bmatrix}^T \mathbf{x}$$
$$= \begin{bmatrix} - & \mathbf{a}_1^T & - \\ & \vdots & \\ - & \mathbf{a}_m^T & - \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^T\mathbf{x} \\ \vdots \\ \mathbf{a}_m^T\mathbf{x} \end{bmatrix} \in \mathbb{R}^m$$

In [68]:
```
A = array([[1, 2],
           [3, 5]])
x = array([-1, 1])
A.transpose() @ x
```

Out[68]:
```
array([2, 3])
```

Matrix-matrix multiplication

- $\mathbf{AB}$: $\mathbf{A}$ multiplied by each column of $\mathbf{B}$

$$\mathbf{AB} = \mathbf{A} \begin{bmatrix} | & & | \\ \mathbf{b}_1 & \cdots & \mathbf{b}_n \\ | & & | \end{bmatrix} = \begin{bmatrix} | & & | \\ \mathbf{Ab}_1 & \cdots & \mathbf{Ab}_n \\ | & & | \end{bmatrix}$$

In [69]:
```
A = array([[1, 2],
           [2, 1]])
B = array([[-1,1],
           [0, 1]])
A @ B
```

Out[69]:
```
array([[-1,  3],
       [-2,  3]])
```

- $\mathbf{A}^T\mathbf{B}$: matrix of inner products between columns of $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{A}^T\mathbf{B} = \mathbf{A}^T \begin{bmatrix} | & & | \\ \mathbf{b}_1 & \cdots & \mathbf{b}_n \\ | & & | \end{bmatrix} = \begin{bmatrix} \mathbf{a_1}^T\mathbf{b_1} & \cdots & \mathbf{a_1}^T\mathbf{b_n} \\ \vdots & \ddots & \vdots \\ \mathbf{a_m}^T\mathbf{b_1} & \cdots & \mathbf{a_m}^T\mathbf{b_n} \end{bmatrix} = \left[\mathbf{a_i}^T\mathbf{b_j}\right]_{ij}$$

In [70]:
```
A = array([[1, 2],
           [2, 1]])
B = array([[-1,1],
```

Out[70]:
```
array([[-1,  3],
       [-2,  3]])
```

- $\mathbf{AB}^T$: sum of outer products of between columns of $\mathbf{A}$ and $\mathbf{B}$ $$\mathbf{A}\mathbf{B}^T = \left[

$$\begin{array}{ccc} | & & | \\ \mathbf{a}_1 & \cdots & \mathbf{a}_n \\ | & & | \end{array}$$

\right] \left[

$$\begin{array}{ccc} - & \mathbf{b}_1^T & - \\ & \vdots & \\ - & \mathbf{b}_n^T & - \end{array}$$

\right] = \sum_{i=1}^n \mathbf{a}_i \mathbf{b}_i^T

$$

In [71]:
```python
A = array([[1, 2],
           [2, 1]])
B = array([[-1,1],
           [0, 1]])
A @ B.transpose()
```

Out[71]:
```
array([[ 1,  2],
       [-1,  1]])
```

Copies and Views

- When operating on arrays, data is sometimes copied and sometimes not.
- *No copy is made for simple assignment.*
  - **Be careful!**

In [72]:
```python
a = array([1,2,3,4])
b = a                   # simple assignment (no copy made!)
b is a                  # yes, b references the same object
```

Out[72]:
```
True
```

In [73]:
```python
b[1] = -2               # changing b also changes a
a
```

Out[73]:
```
array([ 1, -2,  3,  4])
```

- View or shallow copy
  - different array objects can share the same data (called a view)
  - happens when slicing

In [74]:
```python
c = a.view()    # create a view of a
c is a          # not the same object
```

Out[74]:
```
False
```

In [75]:
```python
c.base is a     # but the data is owned by a
```

Out[75]:
```
True
```

In [76]:
```python
c.shape = 2,2   # change shape of c
c
```

Out[76]:
```
array([[ 1, -2],
       [ 3,  4]])
```

In [77]:
```
a                # but the shape of a is the same
```

Out[77]:
```
array([ 1, -2,  3,  4])
```

- Deep copy

In [78]:
```
d = a.copy()         # create a complete copy of a (new data is created)
d is a               # not the same object
```

Out[78]:
```
False
```

In [79]:
```
d.base is a          # not sharing the same data
```

Out[79]:
```
False
```

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. Control structures (conditional and loops)
4. Functions, Classes
5. File IO, Pickle, pandas
6. NumPy
7. **matplotlib**
8. probability review

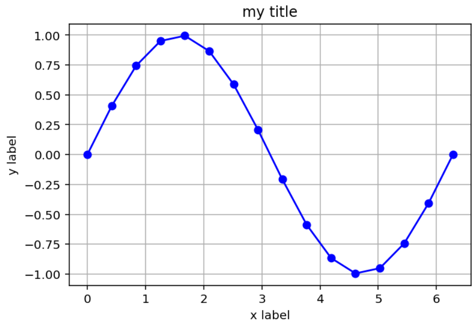
Visualizing Data

- Use matplotlib package to make plots and graphs
- Works with Jupyter to show plots within the notebook

In [80]:
```
# setup matplotlib
%matplotlib inline
# setup output image format (Chrome works best)
import matplotlib_inline   # setup output image format
matplotlib_inline.backend_inline.set_matplotlib_formats('retina')
import matplotlib.pyplot as plt
```

- Each cell will start a new figure automatically.
- Plots are made piece by piece.

In [81]:
```
x = linspace(0,2*pi,16)
y = sin(x)
plt.plot(x, y, 'bo-')
plt.grid(True)
plt.ylabel('y label'); plt.xlabel('x label'); plt.title('my title')
plt.show()
```



- plot string specifies three things (e.g., `'bo-'`)
  - colors:
    - **b**lue, **r**ed, **g**reen, **m**agenta, **c**yan, **y**ellow, blac**k**, **w**hite
  - markers:
    - `'.'` point; `'o'` circle
    - `'+'` plus; `'x'` x
    - `'*'` star; `'s'` square
    - `'v'` triangle down; `'^'` triangle up
    - `'<'` triangle left; `'>'` triangle right
    - `'p'` pentagon; `'8'` octagon;
    - `'h'` hexagon; `'d'` thin_diamond
  - line styles:
    - `'-'` solid line
    - `'--'` dashed line
    - `'-.'` dash-dotted line
    - `':'` dotted lione

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. Control structures (conditional and loops)
4. Functions, Classes
5. File IO, Pickle, pandas
6. NumPy
7. matplotlib
8. **probability review**

Brief Review of Probability

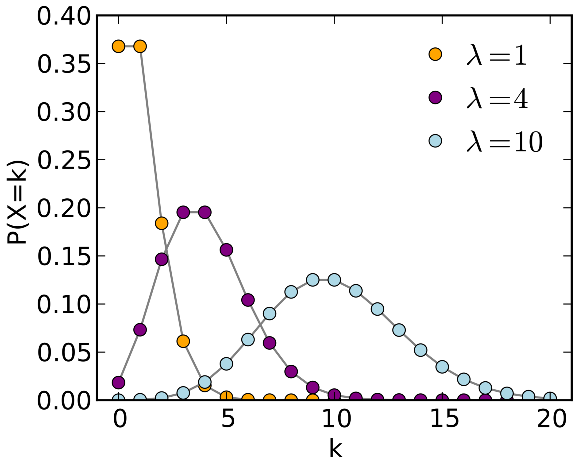
- Random variable (r.v.) X takes a value in $\mathcal{X}$ (set of possible values) at random.
- Associated with a probability distribution $p(X)$ that describes the frequency of outcomes of the X.

Discrete random variables

- Probability mass function (pmf)
- $p(X = x)$ is the probability of r.v. X taking value x
  - we will use simpler notation $p(x)$
- properties
  - $0 \leq p(x) \leq 1$
  - $\sum_{x\in\mathcal{X}} p(x) = 1$ ⇒ "normalized to 1"

- Example: Bernoulli (coin flip)
  - $\mathcal{X} = \{0, 1\}$
  - probability mass function (pmf)
    - $p(x = 1) = \pi \Rightarrow$ "probability of 1 occurring"
    - $p(x = 0) = 1 - \pi \Rightarrow$ "probability of 0 occurring"
    - combined: $p(x) = \pi^x (1 - \pi)^{1-x}$

- Example: Poisson
  - number of arrivals over a fixed time period (e.g., number of phone calls in a fixed interval)
  - $\mathcal{X} = \{0, 1, 2, \cdots\}$
  - λ = average arrival rate ($\lambda > 0$)
  - probability mass function
    - $p(x) = \frac{1}{x!} e^{-\lambda} \lambda^x$



Continuous random variables

- probability density function (pdf).
- $p(x)$ is the likelihood of x.
- properties:
  - $0 \leq p(x) \Rightarrow$ non-negative likelihood
  - $\int p(x)dx = 1$, $\Rightarrow$ "normalized to 1"
  - $p(a \leq x \leq b) = \int_a^b p(x)dx \Rightarrow$ "probability of x between [a,b]"

- Example: Gaussian (Normal)
  - $\mathcal{X} = \mathbb{R}$ (real numbers)
  - μ=mean, σ^2 = variance
    - σ = standard deviation ("the spread of the values")
  - pdf: $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$

Joint probability

- Distribution of more than one r.v.
  - $p(X=x, Y=y)$ - probability that X=x **and** Y=y.
    - simpler notation $p(x,y)$.
- Example:
  - joint probability table (sums to 1)

p(x,y)	Y=0	Y=1
X=0	0.08	0.12
X=1	0.32	0.48

Marginal probability

- Distribution over one r.v. of the joint distribution
- Obtained by summing over the other r.v.
  - Discrete: $p(x) = \sum_{y \in \mathcal{Y}} p(x,y)$
  - Continuous: $p(x) = \int p(x,y)dy$
- Example:

p(x,y)	Y=0	Y=1	**p(x)**
X=0	0.08	0.12	**0.20**
X=1	0.32	0.48	**0.80**
p(y)	**0.40**	**0.60**	

Conditional probability

- Distribution of one r.v. when the value of another r.v. is known (**given**).
  - $p(x|y) = \frac{p(x,y)}{p(y)}$
  - the value y is "given".
- Example:
  - $p(x=0|y=0) = \frac{p(x=0,y=0)}{p(y=0)} = \frac{0.08}{0.4} = 0.2$
  - $p(x=1|y=0) = \frac{p(x=1,y=0)}{p(y=0)} = \frac{0.32}{0.4} = 0.8$
  - $p(x|y=0)$ is a distribution over x, so sums to 1.

Bayes' Rule

- joint probabability can be rewritten as:
  - $p(x,y) = p(x|y)p(y)$
  - $p(x,y) = p(y|x)p(x)$
- Thus,
  - $p(y|x)p(x) = p(x|y)p(y)$
  - $p(y|x) = \frac{p(x|y)p(y)}{p(x)}$

- Looking at denominator...
  - marginalize: $p(x) = \int p(x,y)dy$

- use conditional probability: $p(x) = \int p(x|y)p(y)dy$

- Bayes' Rule
  - $p(y|x) = \frac{p(x|y)p(y)}{\int p(x|y)p(y)dy}$
  - Given only $p(x|y)$ and $p(y)$, we can "invert" the conditioning to obtain $p(y|x)$.
- We will use this next week to build a classifier using probability distributions.

Python Tutorials

- Python - https://docs.python.org/3/tutorial/
- numpy - https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
- “Machine Learning in Action” – Appendix A, Ch. 1
- scikit-learn - http://scikit-learn.org/stable/tutorial/
- matplotlib - http://matplotlib.org/users/pyplot_tutorial.html
- pandas - https://pandas.pydata.org/pandas-docs/stable/tutorials.html