

Introduction to Machine Learning and Data Mining Lecture-2: Python Tools and Math Primer

Prof. Eugene Chang

Today

- Python environments
 - Anaconda, Spyder, IPython, Review
- Python math tools
 - Numpy
- Math Primer
 - Probability and Statistics
 - Bayes rule
 - Linear Algebra
 - Matrix Multiplication and Matrix Inversion
 - Calculus
 - Vector Calculus and Optimization

Software Written in Python

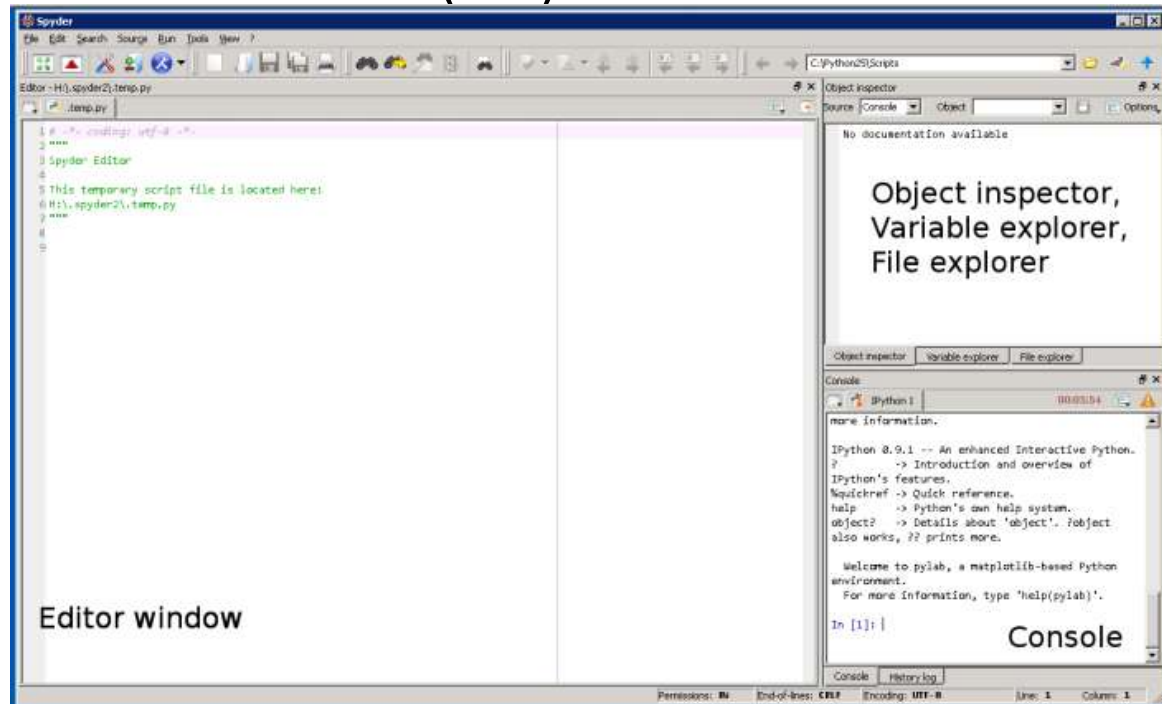
- See http://en.wikipedia.org/wiki/List_of_Python_software
- BitTorrent
- Blender
- Dropbox
- OpenStack
- Large Python projects
 - YouTube
 - Reddit
- Python is also used by Google, Facebook, LucasFilms, NASA, etc.

Anaconda Introduction

- Installation
 - Download and run installer
 - After install
 - conda update conda
 - conda update anaconda
- Where to start
 - Command line shell
 - Launcher: Spyder, Ipython console, and Ipython notebook
- Relevant libraries
 - Numpy (<http://www.numpy.org/>)
 - Scikit-Learn <http://scikit-learn.org/stable/>
 - SciPy (<http://www.scipy.org/>)
 - Matplotlib (<http://matplotlib.org/>)

Spyder

- Comes with Anaconda
 - Windows Shortcut Target (typical installation): C:\Anaconda\pythonw.exe C:\Anaconda\Scripts\spyder-script.py
- Integrated Development Environment (IDE)
 - Code debuggers
 - Information display
 - IPython console



IPython

- Included with Anaconda distributions and available separately
- A much improved command line shell
 - OS and program execution access
 - Object introspection and help
- Great Qt based graphical shell
 - Plotting using pylab
- Browser based **notebook** interface
 - Includes text, code, and graphics
- Examples
 - <https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks>

learn from this git

Examples

- Walk through “Introduction to Python for Econometrics, Statistics and Data Analysis” by Kevin Sheppard
- Walk through notebooks by Kevin Sheppard
 - Class3.ipynb
- All materials are available from SVU FTP:
http://class.svuca.edu/~eugene.chang/class/CS596-029_2015_Summer/

- Example code

(Class 3)

$x = [1, 2]$

$y = x[:]$ ^{slice & all} # clone data from x instead of reference

$y = x[start : end : incremental]$

using command line 'launcher' and.
launch ipython-notebook app.

Review Character Strings

Dynamic typing – no declaration

No memory allocation

Immutable

```
s = "Good Afternoon"
```

```
len(s)
```

length of string

Review String Slicing

```
s = "Good Afternoon"
```

```
s[0] evaluates to "G"
```

inclusive → *exclusive*

```
s[5:10] selects "After" # string slicing
```

```
s[:10] selects "Good After"
```

```
s[5:] selects "Afternoon"
```

```
s[-4:] selects "noon" # last 4 characters
```

↑
go backward.

String Methods

String is a Class with data & subroutines:

```
t = s.upper()  
pos = s.find("A")
```

```
first = "George"  
last = "Washington"  
name = first + " " + last  
# string concatenation
```

Review Lists

Ordered sequence of items

Can be floats, ints, strings, Lists

Mutable

```
a = [16, 25.3, "hello", 45]
```

```
a[0] contains 16
```

```
a[-1] contains 45
```

```
a[0:2] is a list containing [16, 25.3]
```

Create a List

```
days = [ ]  
days.append("Monday")  
days.append("Tuesday")  
  
years = range(2000, 2014)
```

List Methods

List is a Class with data & subroutines:

d.insert()

d.remove()

d.sort()

Can concatenate lists with +

String split

```
s = "Machine Learning Data Mining"
```

```
myList = s.split() # returns a list of strings
```

↑ can use different delimiter

```
print myList
```

```
[ "Machine", "Learning", "Data", "Mining" ]
```

```
help(str.split) # delimiters, etc.
```

Tuple

Designated by () parenthesis

A List that can not be changed. Immutable.
No append.

Good for returning multiple values from a subroutine function.

Can extract slices.

Review math module

1st way of using external lib
`import math`
`dir(math)`

`math.sqrt(x)`
`math.sin(x)`
`math.cos(x)`

2nd way
`from math import *`
`dir()`

`sqrt(x)`

3rd way
`from math import pi`
`dir()`

`print pi`

import a module

```
import math                # knows where to find it
```

```
import sys  
sys.path.append("C:\Users\Yuhlin\Documents\Python Scripts")
```

```
import mpython.py         # import your own code
```

```
if task == 3:  
    import math            # imports can be anywhere
```

another way is to define the PATH in Spyder Path Manager

Review Defining a Function

Block of code separate from main.

Define the function before calling it.

```
def myAdd(a, b):           # define before calling
    return a + b
```

```
p = 25                     # main section of code
q = 30
```

```
r = myAdd(p, q)
```

Keyword Arguments

Provide default values for optional arguments.

```
def setLineAttributes(color="black",  
    style="solid", thickness=1):  
    ...
```

Call function from main program

```
setLineAttributes(style="dotted")  
setLineAttributes("red", thickness=2)
```

NumPy

Travis E. Oliphant
oliphant@enthought.com

Enthought, Inc.
www.enthought.com

What is NumPy?

- Python is a fabulous language
 - Easy to extend
 - Great syntax which encourages easy to write and maintain code
 - Incredibly large standard-library and third-party tools
- **No built-in multi-dimensional array** (but it supports the needed syntax for extracting elements from one)
- NumPy provides a **fast** built-in object (ndarray) which is a multi-dimensional array of a homogeneous data-type.

NumPy

- Website <http://numpy.scipy.org/>
- Offers Matlab-ish capabilities within Python
- NumPy replaces Numeric and Numarray
- Initially developed by Travis Oliphant (building on the work of others)
- NumPy 1.0 released October, 2006, current version 1.9.2

Overview of NumPy

N-D ARRAY (NDARRAY)

- N-dimensional array of rectangular data
- Element of the array can be C-structure or simple data-type.
- Fast algorithms on machine data-types (int, float, etc.)

fixed data type

v/s list is freestyle

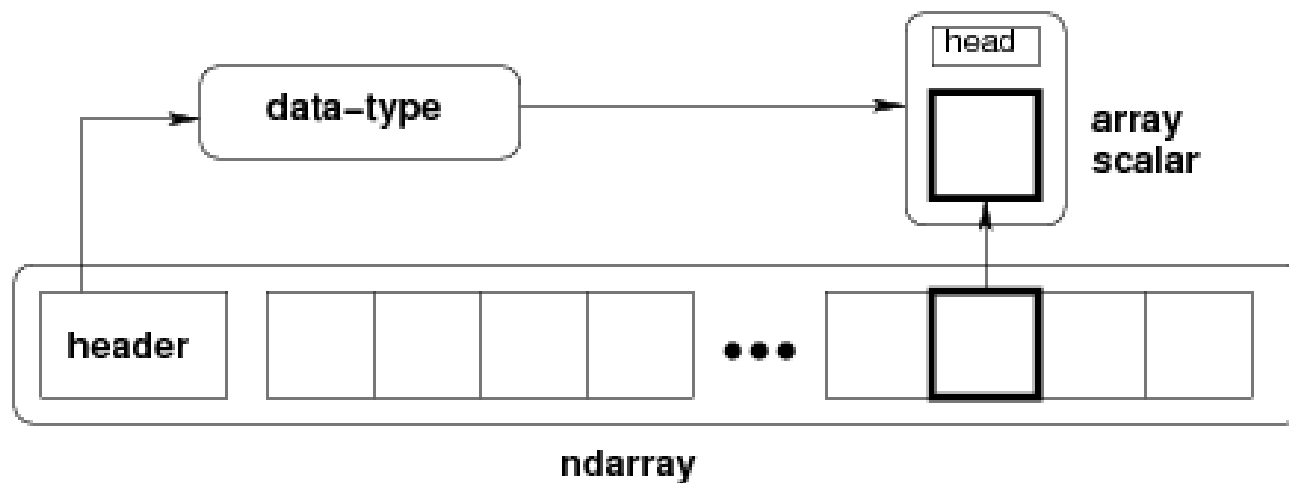
UNIVERSAL FUNCTIONS (UFUNC)

- functions that operate element-by-element and return result
- fast-loops registered for each fundamental data-type
 - $\sin(x) = [\sin(x_i) \text{ } i=0..N]$
 - $x+y = [x_i + y_i \text{ } i=0..N]$

Func works on all elements

NumPy Array

A NumPy array is an N-dimensional homogeneous collection of “items” of the same “kind”. The kind can be any arbitrary structure and is specified using the data-type.



NumPy Array

A NumPy array is a homogeneous collection of “items” of the same “data-type” (dtype)

```
>>> import numpy as N
>>> a =
N.array([[1,2,3],[4,5,6]],float)
>>> print a
[[1. 2. 3.]
 [4. 5. 6.]]
>>> print a.shape, "\n", a.itemsize
(2, 3)
8
>>> print a.dtype, a.dtype.type
'<f8' <type 'float64scalar'>
>>> type(a[0,0])
<type 'float64scalar'>
>>> type(a[0,0]) is type(a[1,2])
True
```

data type of the N-d.
↓

Introducing NumPy Arrays

SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
```

CHECKING THE TYPE

```
>>> type(a)
<type 'array'>
```

NUMERIC 'TYPE' OF ELEMENTS

```
>>> a.dtype
dtype('int32')
```

BYTES PER ELEMENT

```
>>> a.itemsize # per element
4
```

ARRAY SHAPE

```
# shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

ARRAY SIZE

```
# size reports the entire
# number of elements in an
# array.
>>> a.size
4
>>> size(a)
4
```

Introducing NumPy Arrays

BYTES OF MEMORY USED

```
# returns the number of bytes
# used by the data portion of
# the array.
>>> a.nbytes
12
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
1
```

ARRAY COPY

```
# create a copy of the array
>>> b = a.copy()
>>> b
array([0, 1, 2, 3])
```

CONVERSION TO LIST

```
# convert a numpy array to a
# python list.
>>> a.tolist()
[0, 1, 2, 3]
```

```
# For 1D arrays, list also
# works equivalently, but
# is slower.
>>> list(a)
[0, 1, 2, 3]
```

Setting Array Elements

ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
[10, 1, 2, 3]
```

FILL

set all values in an array.

```
>>> a.fill(0)
>>> a
[0, 0, 0, 0]
```

This also works, but may
be slower.

```
>>> a[:] = 1
>>> a
[1, 1, 1, 1]
```



BEWARE OF TYPE COERSION

```
>>> a.dtype
dtype('int32')
```

assigning a float to into # an int32 array
will

truncate decimal part.

```
>>> a[0] = 10.6
>>> a
[10, 1, 2, 3]
```

fill has the same behavior

```
>>> a.fill(-4.8)
>>> a
[-4, -4, -4, -4]
```

Multi-Dimensional Arrays

MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

(ROWS,COLUMNS)

```
>>> a.shape
(2, 4)
>>> shape(a)
(2, 4)
```

ELEMENT COUNT

```
>>> a.size
8
>>> size(a)
8
```

NUMBER OF DIMENSIONS

```
>>> a.ndim
2
```

GET/SET ELEMENTS

```
>>> a[1,3]
13
```



```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

ADDRESS FIRST ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, -1])
```

Array Slicing

SLICING WORKS MUCH LIKE
STANDARD PYTHON SLICING

only 0th row
`>>> a[0,3:5]`
`array([3, 4])`
slice on that row

4 to end row
`>>> a[4:,4:]`
`array([[44, 45],`
`[54, 55]])`

select 2nd col of each row
`>>> a[:,2]`
`array([2,12,22,32,42,52])`

STRIDES ARE ALSO POSSIBLE

```
>>> a[2::2,:2]
array([[20, 22, 24],
      [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Memory Model

```
>>> print a.strides
(24, 8)
>>> print a.flags.fortran, a.flags.contiguous
False True
>>> print a.T.strides
(8, 24)
>>> print a.T.flags.fortran, a.T.flags.contiguous
True False
```

- Every dimension of an ndarray is accessed by stepping (striding) a fixed number of bytes through memory.
- If memory is contiguous, then the strides are “pre-computed” indexing-formulas for either Fortran-order (first-dimension varies the fastest), or C-order (last-dimension varies the fastest) arrays.

Array slicing (Views)

Memory model allows “simple indexing” (integers and slices) into the array to be a **view** of the same data.

```
>>> b = a[:, ::2]
>>> b[0,1] = 100
>>> print a
[[  1.    2.  100.]]
[  4.    5.    6.]]
>>> c =
a[:, ::2].copy()
>>> c[1,0] = 500
>>> print a
[[  1.    2.  100.]]
[  4.    5.    6.]]
```

Handwritten notes: "create copy" with an arrow pointing to `a[:, ::2].copy()` and "no reference" with an arrow pointing to `b = a[:, ::2]`.

Other uses of view

```
>>> b = a.view('i8')
>>> [hex(val.item()) for
val in b.flat]
['0x3FF0000000000000L',
'0x4000000000000000L',
'0x4059000000000000L',
'0x4010000000000000L',
'0x4014000000000000L',
'0x4018000000000000L']
```


Slices Are References

Slices are references to memory in original array. Changing values in a slice also changes the original array.

```
>>> a = array([0,1,2,3,4])
```

```
# create a slice containing only the  
# last element of a
```

```
>>> b = a[2:4]
```

```
>>> b[0] = 10
```

```
# changing b changed a!
```

```
>>> a
```

```
array([ 1, 2, 10, 3, 4])
```

| different from Python list slicing
| which CLONES the data

Fancy Indexing

← for numpy array only

INDEXING BY POSITION

```
>>> a = arange(0,80,10)
```

```
# fancy indexing
```

```
>>> y = a[[1, 2, -3]]
```

```
>>> print y
```

```
[10 20 50]
```

```
# using take
```

```
>>> y = take(a,[1,2,-3])
```

```
>>> print y
```

```
[10 20 50]
```

← only apply for numpy array.
arange
populates
the array
content.

INDEXING WITH BOOLEANS

```
>>> mask = array([0,1,1,0,0,1,0,0],  
...              dtype=bool)
```

```
# fancy indexing
```

```
>>> y = a[mask]
```

```
>>> print y
```

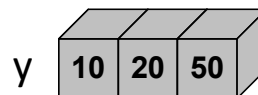
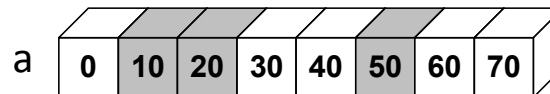
```
[10,20,50]
```

```
# using compress
```

```
>>> y = compress(mask, a)
```

```
>>> print y
```

```
[10,20,50]
```



Fancy Indexing in 2D

```
>>> a[(0,1,2,3,4),(1,2,3,4,5)]  
array([ 1, 12, 23, 34, 45])
```

```
>>> a[3:,[0, 2, 5]]  
array([[30, 32, 35],  
       [40, 42, 45]]  
       [50, 52, 55]])
```

```
>>> mask = array([1,0,1,0,0,1],  
                 dtype=bool)  
>>> a[mask,2]  
array([2,22,52])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55



Unlike slicing, fancy indexing creates copies instead of views into original arrays.

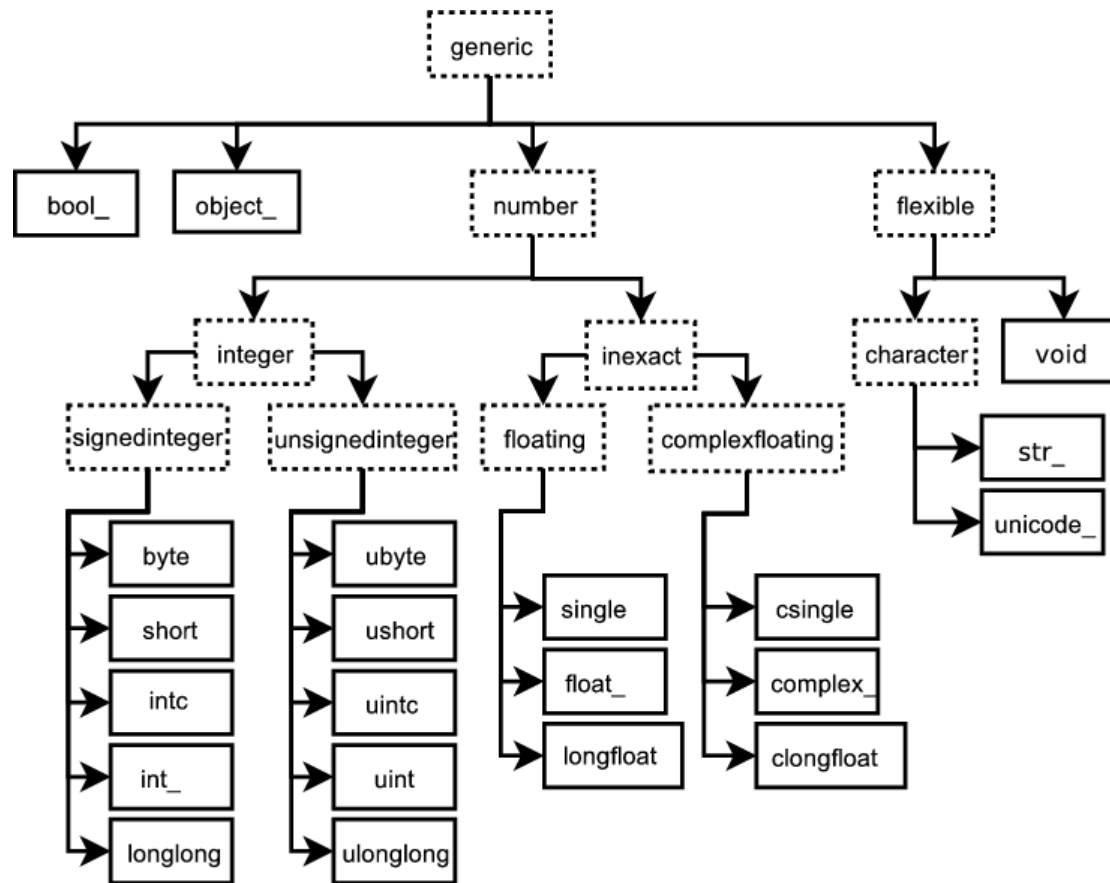
Data-types

- There are two related concepts of “type”
 - The data-type object (dtype)
 - The Python “type” of the object created from a single array item (hierarchy of scalar types)
- The **dtype** object provides the details of how to interpret the memory for an item. It's an instance of a single dtype class.
- The “type” of the extracted elements are true Python classes that exist in a hierarchy of Python classes
- Every dtype object has a type attribute which provides the Python object returned when an element is selected from the array

NumPy dtypes

Basic Type	Available NumPy types	Comments
Boolean	<code>bool</code>	Elements are 1 byte in size
Integer	<code>int8, int16, int32, int64, int128, int</code>	<code>int</code> defaults to the size of <code>int</code> in C for the platform
Unsigned Integer	<code>uint8, uint16, uint32, uint64, uint128, uint</code>	<code>uint</code> defaults to the size of unsigned <code>int</code> in C for the platform
Float	<code>float32, float64, float, longfloat,</code>	Float is always a double precision floating point value (64 bits). <code>longfloat</code> represents large precision floats. Its size is platform dependent.
Complex	<code>complex64, complex128, complex</code>	The real and complex elements of a <code>complex64</code> are each represented by a single precision (32 bit) value for a total size of 64 bits.
Strings	<code>str, unicode</code>	Unicode is always UTF32 (UCS4)
Object	<code>object</code>	Represent items in array as Python objects.
Records	<code>void</code>	Used for arbitrary data structures in record arrays.

Built-in “scalar” types



Data-type object (dtype)

- There are 21 “built-in” (static) data-type objects
- New (dynamic) data-type objects are created to handle
 - Alteration of the byteorder
 - Change in the element size (for string, unicode, and void built-ins)
 - Addition of fields
 - Change of the type object (C-structure arrays)
- Creation of data-types is quite flexible.
- New user-defined “built-in” data-types can also be added (but must be done in C and involves filling a function-pointer table)

Data-type fields

- An item can include fields of different data-types.
- A field is described by a data-type object and a byte offset --- this definition allows nested records.
- The array construction command interprets tuple elements as field entries.

```
>>> dt = N.dtype("i4,f8,a5")
>>> print dt.fields
{'f1': (dtype('<i4'), 0), 'f2': (dtype('<f8'), 4),
'f3': (dtype('|S5'), 12)}
>>> a = N.array([(1,2.0,"Hello"), (2,3.0,"World")],
dtype=dt)
>>> print a['f3']
[Hello World]
```


Array Calculation Methods

SUM FUNCTION

```
>>> a = array([[1,2,3], [4,5,6]], float)
```

```
# Sum defaults to summing all  
# *all* array values.
```

```
>>> sum(a)  
21.
```

```
# supply the keyword axis to  
# sum along the 0th axis.
```

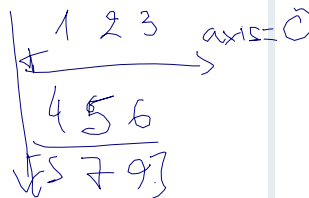
```
>>> sum(a, axis=0)  
array([5., 7., 9.])
```

axis=0: Sum across dimension 0

```
# supply the keyword axis to  
# sum along the last axis.
```

```
>>> sum(a, axis=-1)  
array([6., 15.])
```

*should try with 3 rows
to see the differences
between +1 or -1*



SUM ARRAY METHOD

```
# The a.sum() defaults to  
# summing *all* array values
```

```
>>> a.sum()  
21.
```

```
# Supply an axis argument to  
# sum along a specific axis.
```

```
>>> a.sum(axis=0)  
array([5., 7., 9.])
```

PRODUCT

```
# product along columns.
```

```
>>> a.prod(axis=0)  
array([ 4., 10., 18.])
```

```
# functional form.
```

```
>>> prod(a, axis=0)  
array([ 4., 10., 18.])
```

Min/Max

amin & amax give better performance result than default python min & max func on multi-D array

MIN

```
>>> a = array([2.,3.,0.,1.]) >>> a.min(axis=0)
0.
# use Numpy's amin() instead
# of Python's builtin min()
# for speed operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.
```

ARGMIN

```
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# functional form
>>> argmin(a, axis=0)
2
```

MAX

```
>>> a = array([2.,1.,0.,3.]) >>> a.max(axis=0)
3.

# functional form
>>> amax(a, axis=0)
3.
```

ARGMAX

```
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# functional form
>>> argmax(a, axis=0)
1
```

Statistics Array Methods

MEAN

```
>>> a = array([[1,2,3],
               [4,5,6]], float)

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
>>> average(a, axis=0)
array([ 2.5,  3.5,  4.5])

# average can also calculate
# a weighted average
>>> average(a, weights=[1,2],
...         axis=0)
array([ 3.,  4.,  5.])
```

STANDARD DEV./VARIANCE

```
# Standard Deviation
>>> a.std(axis=0)
array([ 1.5,  1.5,  1.5])

# Variance
>>> a.var(axis=0)
array([2.25, 2.25, 2.25])
>>> var(a, axis=0)
array([2.25, 2.25, 2.25])
```

Other Array Methods

CLIP

Limit values to a range

```
>>> a = array([[1,2,3],  
               [4,5,6]], float)
```

Set values < 3 equal to 3.

Set values > 5 equal to 5.

```
>>> a.clip(3,5)
```

```
>>> a
```

```
array([[ 3.,  3.,  3.],  
       [ 4.,  5.,  5.]])
```

ROUND

Round values in an array.

Numpy rounds to even, so

1.5 and 2.5 both round to 2.

*weird numpy
rounding.*

```
>>> a = array([1.35, 2.5, 1.5])
```

```
>>> a.round()
```

```
array([ 1.,  2.,  2.])
```

Round to first decimal place.

```
>>> a.round(decimals=1)
```

```
array([ 1.4,  2.5,  1.5])
```

POINT TO POINT

Calculate max – min for

array along columns

```
>>> a.ptp(axis=0)
```

```
array([ 3.0,  3.0,  3.0])
```

max – min for entire array.

```
>>> a.ptp(axis=None)
```

```
5.0
```

Summary of (most) array attributes/methods

BASIC ATTRIBUTES

`a.dtype` - Numerical type of array elements. float32, uint8, etc.
`a.shape` - Shape of the array. (m,n,o,...)
`a.size` - Number of elements in entire array.
`a.itemsize` - Number of bytes used by a single element in the array.
`a.nbytes` - Number of bytes used by entire array (data only).
`a.ndim` - Number of dimensions in the array.

SHAPE OPERATIONS

`a.flat` - An iterator to step through array as if it is 1D.
`a.flatten()` - Returns a 1D copy of a multi-dimensional array.
`a.ravel()` - Same as flatten(), but returns a 'view' if possible.
`a.resize(new_size)` - Change the size/shape of an array in-place.
`a.swapaxes(axis1, axis2)` - Swap the order of two axes in an array.
`a.transpose(*axes)` - Swap the order of any number of array axes.
`a.T` - Shorthand for `a.transpose()`
`a.squeeze()` - Remove any length=1 dimensions from an array.

Summary of (most) array attributes/methods

FILL AND COPY

`a.copy()` - Return a copy of the array.
`a.fill(value)` - Fill array with a scalar value.

CONVERSION / COERSION

`a.tolist()` - Convert array into nested lists of values.
`a.tostring()` - raw copy of array memory into a python string.
`a.astype(dtype)` - Return array coerced to given dtype.
`a.byteswap(False)` - Convert byte order (big <-> little endian).

COMPLEX NUMBERS

`a.real` - Return the real part of the array.
`a.imag` - Return the imaginary part of the array.
`a.conjugate()` - Return the complex conjugate of the array.
`a.conj()` - Return the complex conjugate of an array. (same as conjugate)

Summary of (most) array attributes/methods

SAVING

`a.dump(file)` - Store a binary array data out to the given file.
`a.dumps()` - returns the binary pickle of the array as a string.
`a.tofile(fid, sep="", format="%s")` Formatted ascii output to file.

SEARCH / SORT

`a.nonzero()` - Return indices for all non-zero elements in a.
`a.sort(axis=-1)` - Inplace sort of array elements along axis.
`a.argsort(axis=-1)` - Return indices for element sort order along axis.
`a.searchsorted(b)` - Return index where elements from b would go in a.

ELEMENT MATH OPERATIONS

`a.clip(low, high)` - Limit values in array to the specified range.
`a.round(decimals=0)` - Round to the specified number of digits.
`a.cumsum(axis=None)` - Cumulative sum of elements along axis.
`a.cumprod(axis=None)` - Cumulative product of elements along axis.

Summary of (most) array attributes/methods

REDUCTION METHODS

All the following methods “reduce” the size of the array by 1 dimension by carrying out an operation along the specified axis. If axis is None, the operation is carried out across the entire array.

`a.sum(axis=None)` - Sum up values along axis.
`a.prod(axis=None)` - Find the product of all values along axis.
`a.min(axis=None)` - Find the minimum value along axis.
`a.max(axis=None)` - Find the maximum value along axis.
`a.argmin(axis=None)` - Find the index of the minimum value along axis.
`a.argmax(axis=None)` - Find the index of the maximum value along axis.
`a.ptp(axis=None)` - Calculate `a.max(axis) - a.min(axis)`
`a.mean(axis=None)` - Find the mean (average) value along axis.
`a.std(axis=None)` - Find the standard deviation along axis.
`a.var(axis=None)` - Find the variance along axis.

`a.any(axis=None)` - True if any value along axis is non-zero. (or)
`a.all(axis=None)` - True if all values along axis are non-zero. (and)

Array Operations

SIMPLE ARRAY MATH

```
>>> a = array([1,2,3,4])
>>> b = array([2,3,4,5])
>>> a + b
array([3, 5, 7, 9])
```



Numpy defines the following constants:

```
pi = 3.14159265359
e  = 2.71828182846
```

MATH FUNCTIONS

```
# Create array from 0 to 10
>>> x = arange(11.)
```

```
# multiply entire array by
# scalar value
```

```
>>> a = (2*pi)/10.
>>> a
0.62831853071795862
>>> a*x
array([ 0.,0.628,...,6.283])
```

```
# inplace operations
```

```
>>> x *= a
>>> x
array([ 0.,0.628,...,6.283])
```

```
# apply functions to array.
```

```
>>> y = sin(x)
```

Universal Functions

- ufuncs are objects that rapidly evaluate a function element-by-element over an array.
- Core piece is a 1-d loop written in C that performs the operation over the largest dimension of the array
- For 1-d arrays it is equivalent to but much faster than list comprehension

```
>>> type(N.exp)
<type 'numpy.ufunc'>
>>> x = array([1,2,3,4,5])
>>> print N.exp(x)
[  2.71828183   7.3890561   20.08553692
 54.59815003  148.4131591 ]
>>> print [math.exp(val) for val in x]
[2.7182818284590451,
 7.3890560989306504, 20.085536923187668,
 54.598150033144236, 148.4131591025766]
```

Mathematic Binary Operators

$a + b \rightarrow \text{add}(a,b)$
 $a - b \rightarrow \text{subtract}(a,b)$
 $a \% b \rightarrow \text{remainder}(a,b)$

MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

ELEMENT BY ELEMENT ADDITION

```
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

$a * b \rightarrow \text{multiply}(a,b)$
 $a / b \rightarrow \text{divide}(a,b)$
 $a ** b \rightarrow \text{power}(a,b)$

*power, double **

ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

IN PLACE OPERATION

```
# Overwrite contents of a.
# Saves array creation
# overhead
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```

Comparison and Logical Operators

<code>equal</code>	<code>(==)</code>	<code>not_equal</code>	<code>(!=)</code>	<code>greater</code>	<code>(>)</code>
<code>greater_equal</code>	<code>(>=)</code>	<code>less</code>	<code>(<)</code>	<code>less_equal</code>	<code>(<=)</code>
<code>logical_and</code>		<code>logical_or</code>		<code>logical_xor</code>	
<code>logical_not</code>					

2D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])
# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```

Bitwise Operators

<code>bitwise_and</code>	<code>(&)</code>	<code>invert</code>	<code>(~)</code>	<code>right_shift(a,shifts)</code>
<code>bitwise_or</code>	<code>()</code>	<code>bitwise_xor</code>		<code>left_shift (a,shifts)</code>

BITWISE EXAMPLES

```
>>> a = array((1,2,4,8))
>>> b = array((16,32,64,128))
>>> bitwise_or(a,b)
array([ 17, 34, 68, 136])
```

bit inversion

```
>>> a = array((1,2,3,4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)
```

left shift operation

```
>>> left_shift(a,3)
array([ 8, 16, 24, 32], dtype=uint8)
```

Trig and Other Functions

TRIGONOMETRIC

<code>sin(x)</code>	<code>sinh(x)</code>
<code>cos(x)</code>	<code>cosh(x)</code>
<code>arccos(x)</code>	<code>arccosh(x)</code>
<code>arctan(x)</code>	<code>arctanh(x)</code>
<code>arcsin(x)</code>	<code>arcsinh(x)</code>
<code>arctan2(x, y)</code>	

OTHERS

<code>exp(x)</code>	<code>log(x)</code>
<code>log10(x)</code>	<code>sqrt(x)</code>
<code>absolute(x)</code>	<code>conjugate(x)</code>
<code>negative(x)</code>	<code>ceil(x)</code>
<code>floor(x)</code>	<code>fabs(x)</code>
<code>hypot(x, y)</code>	<code>fmod(x, y)</code>
<code>maximum(x, y)</code>	<code>minimum(x, y)</code>

hypot(x,y)

Element by element distance
calculation using $\sqrt{x^2 + y^2}$

Broadcasting

When there are multiple inputs, then they all must be “broadcastable” to the same shape.

- All arrays are promoted to the same number of dimensions (by prepending 1's to the shape)
- All dimensions of length 1 are expanded as determined by other inputs with non-unit lengths in that dimension.

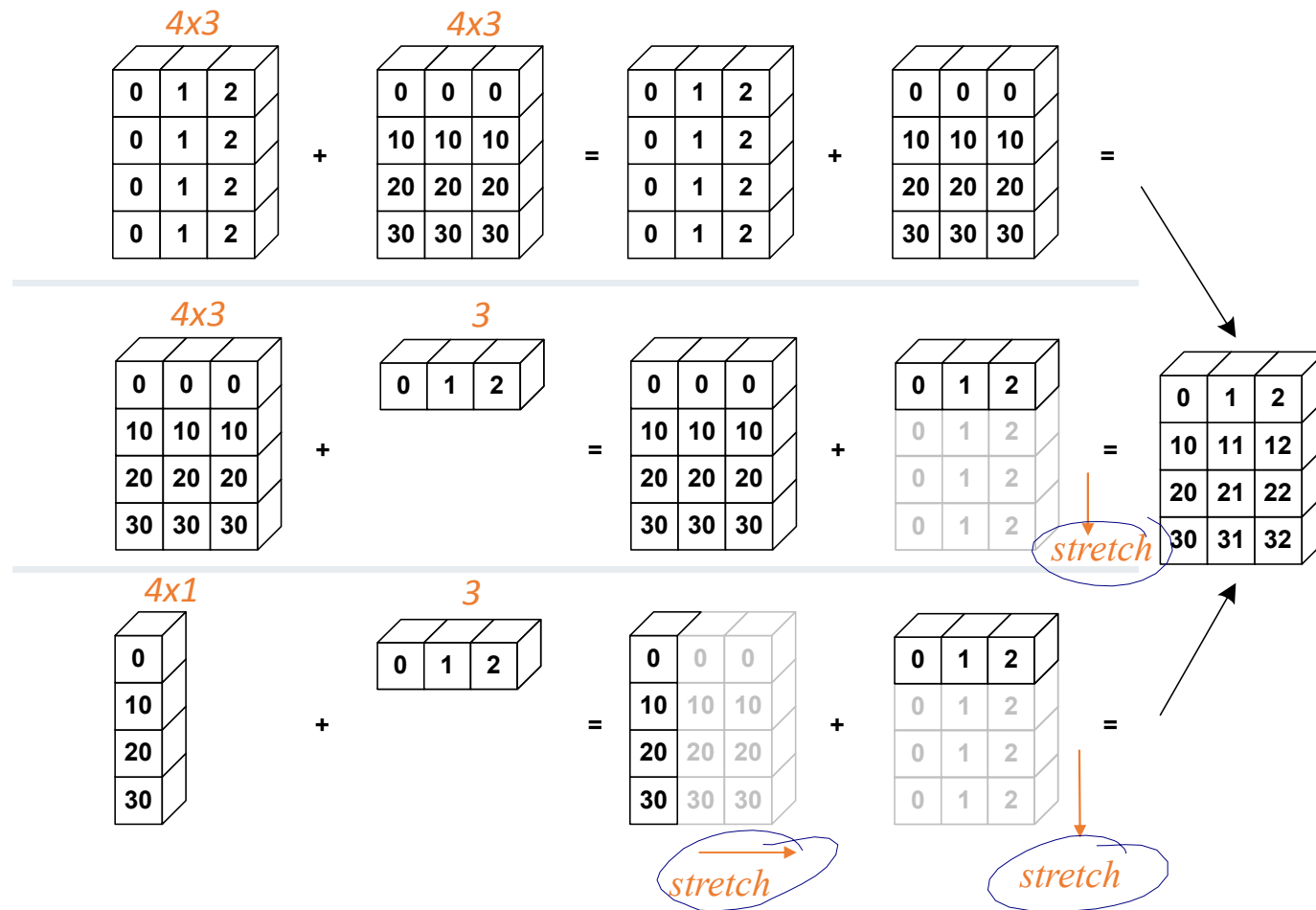
```
>>> x = [1,2,3,4];  
>>> y =  
[[10],[20],[30]]  
>>> print N.add(x,y)  
[[11 12 13 14]  
 [21 22 23 24]  
 [31 32 33 34]]  
>>> x = array(x)  
>>> y = array(y)  
>>> print x+y  
[[11 12 13 14]  
 [21 22 23 24]  
 [31 32 33 34]]
```

x has shape (4,) the ufunc
sees it as having shape (1,4)

y has shape (3,1)

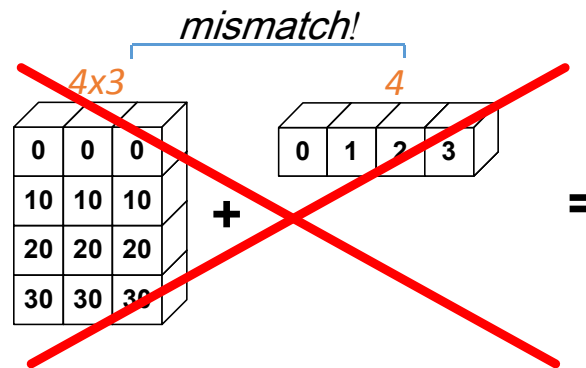
The ufunc result has shape
(3,4)

Array Broadcasting



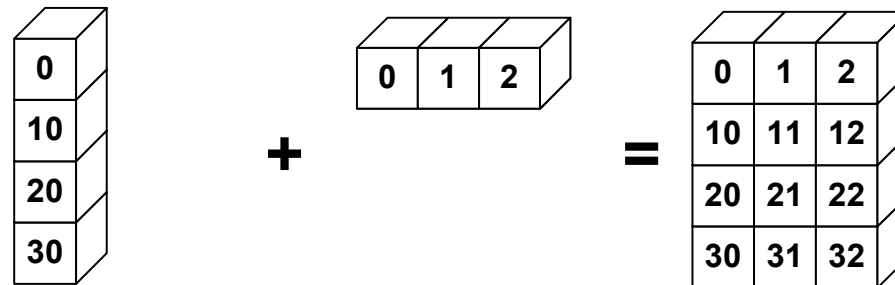
Broadcasting Rules

The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur. Otherwise, a `ValueError: frames are not aligned` exception is thrown.



Broadcasting in Action

```
>>> a = array((0,10,20,30))  
>>> b = array((0,1,2))  
>>> y = a[:, None] + b
```



Universal Function Methods

The mathematic, comparative, logical, and bitwise operators that take two arguments (binary operators) have special methods that operate on arrays:

```
op.reduce(a, axis=0)
```

```
op.accumulate(a, axis=0)
```

```
op.outer(a, b)
```

```
op.reduceat(a, indices)
```

Vectorizing Functions

VECTORIZING FUNCTIONS

Example

```
# special.sinc already available
# This is just for show.
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

SOLUTION

```
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc([1.3,1.5])
array([-0.1981, -0.2122])
```

attempt

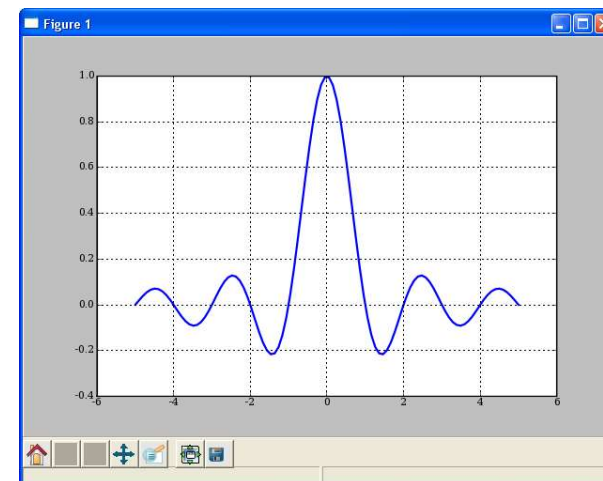
```
>>> sinc([1.3,1.5])
```

TypeError: can't multiply sequence to non-int

```
>>> x = r_[-5:5:100j]
```

```
>>> y = vsinc(x)
```

```
>>> plot(x, y)
```



Random Number Generation

- The Random Seed

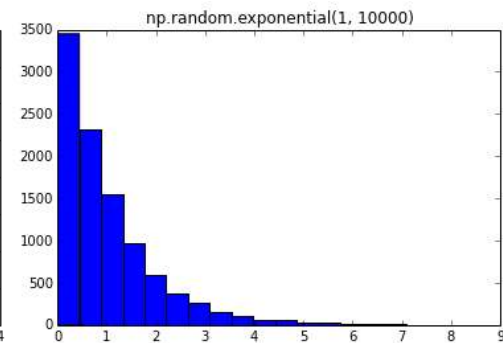
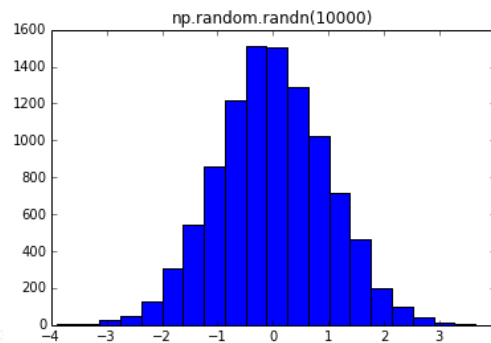
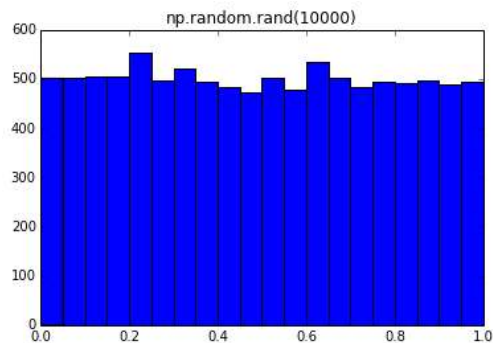
```
>>> np.random.seed(10)
```

- Various Distributions

```
>>> np.random.SOME_DISTRIBUTION(params, size=(dimensions))
```

```
>>> # rand ~ Uniform; randn ~ Normal; exponential; beta;
```

```
>>> # gamma; binomial; randint(=>Discrete) ... even triangular
```



Numerical Linear Algebra

- Matrix Decompositions

```
>>> A = np.ones((5,5)) + np.eye(5)
>>> L = np.linalg.cholesky(A)                # Cholesky decomposition.
>>> np.allclose(np.dot(L,L.T), A) # Check...
True
>>> Q, R = np.linalg.qr(A)                   # QR decomposition
>>> np.allclose(np.dot(Q,R), A)
True
>>> # Eigenvalue decomposition (Use eigh when symmetric/Hermetian)
>>> e, V = np.linalg.eig(A)
>>> np.allclose(np.dot(A,V), np.dot(V,np.diag(e))) # V may not be unitary
True
>>> # Singular Value Decomposition
>>> U, s, V = np.linalg.svd(A, full_matrices = True)
>>> np.allclose(np.dot(np.dot(U, np.diag(s)),V), A)
True
```

Numerical Linear Algebra

- Solving Linear Systems

```
>>> np.linalg.solve(A, np.ones((5,1))).T    # Solve Linear System
array([[ 0.16666667,  0.16666667,  0.16666667,  0.16666667,  0.16666667]])

>>> np.linalg.inv(A)                        # Matrix Inverse
array([[ 0.83333333, -0.16666667, -0.16666667, -0.16666667, -0.16666667],
       [-0.16666667,  0.83333333, -0.16666667, -0.16666667, -0.16666667],
       [-0.16666667, -0.16666667,  0.83333333, -0.16666667, -0.16666667],
       [-0.16666667, -0.16666667, -0.16666667,  0.83333333, -0.16666667],
       [-0.16666667, -0.16666667, -0.16666667, -0.16666667,  0.83333333]])
```

- Other Stuff

```
>>> np.linalg.norm(A)                      # Matrix norm
6.324555320336759

>>> np.linalg.det(A)                       # Determinant (should be 6...)
5.9999999999999982

>>> np.trace(A)                            # Matrix trace
10

>>> np.linalg.matrix_rank(A)               # Matrix rank
5
```

Math Primer

Math Tools for Machine Learning

- Statistics and Probabilities
 - Modeling
- Calculus
 - Optimization: need to identify the maximum likelihood, or minimum risk.
 - Integration allows the marginalization of continuous probability density functions
- Linear Algebra
 - Many features leads to high dimensional spaces
 - Vectors and matrices allow us to compactly describe and manipulate high dimensional feature spaces. *filter & reduce the data*

Linear Algebra

- Vectors

- A one dimensional array.
- If not specified, assume x is a column vector.

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix}$$

- Matrices *2d*

- Higher dimensional array.
- Typically denoted with capital letters.
- n rows by m columns

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & & a_{1,m-1} \\ \vdots & & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \dots & a_{n-1,m-1} \end{pmatrix}$$

Transposition

- **Transposing** a matrix swaps columns and rows.

$$\mathbf{x} = \begin{pmatrix} x_0 \\ x_1 \\ \dots \\ x_{n-1} \end{pmatrix}$$

$$\mathbf{x}^T = (x_0 \quad x_1 \quad \dots \quad x_{n-1})$$

Transposition

- **Transposing** a matrix swaps columns and rows.

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & & a_{1,m-1} \\ \vdots & & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \dots & a_{n-1,m-1} \end{pmatrix}$$
$$A^T = \begin{pmatrix} a_{0,0} & a_{1,0} & \dots & a_{n-1,0} \\ a_{0,1} & a_{1,1} & & a_{1,m-1} \\ \vdots & & \ddots & \vdots \\ a_{0,m-1} & a_{1,m-1} & \dots & a_{n-1,m-1} \end{pmatrix}$$

Addition

- Matrices can be added to themselves iff they have the same dimensions.
 - A and B are both n-by-m matrices.

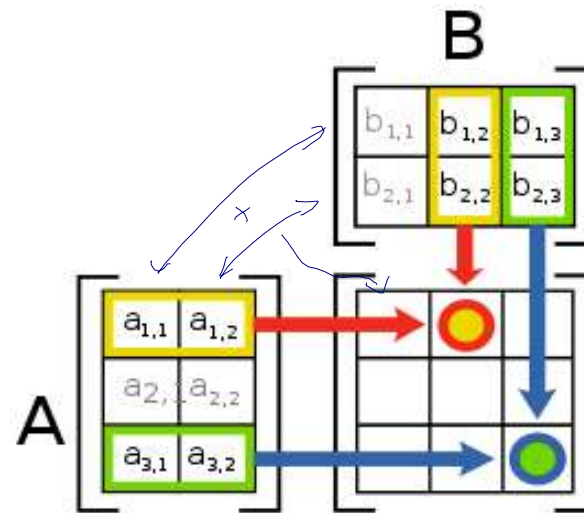
$$A + B = \begin{pmatrix} a_{0,0} + b_{0,0} & a_{0,1} + b_{0,1} & \dots & a_{0,m-1} + b_{0,m-1} \\ a_{1,0} + b_{1,0} & a_{1,1} + b_{1,1} & & a_{1,m-1} + b_{1,m-1} \\ \vdots & & \ddots & \vdots \\ a_{n-1,0} + b_{n-1,0} & a_{n-1,1} + b_{n-1,1} & \dots & a_{n-1,m-1} + b_{n-1,m-1} \end{pmatrix}$$

Multiplication

- To multiply two matrices, the **inner dimensions** must be the same.
 - An n -by- m matrix can be multiplied by an m -by- k matrix

$$AB = C$$

$$c_{ij} = \sum_{k=0}^m a_{ik} * b_{kj}$$



$$c_{00} = a_{00} \times b_{00} + a_{01} \times b_{10}$$

$$c_{01} = a_{00} \times b_{01} + a_{01} \times b_{11}$$

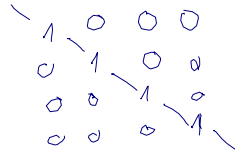
$$c_{02} = a_{00} \times b_{02} + a_{01} \times b_{12} + a_{02} \times b_{22}$$

Inversion

- The inverse of an n-by-n or **square** matrix A is denoted A^{-1} , and has the following property.

$$AA^{-1} = \bar{I} \quad \text{identity matrix.}$$

- Where I is the **identity** matrix is an n-by-n matrix with ones along the diagonal.
 - $I_{ij} = 1$ iff $i = j$, 0 otherwise



Identity Matrix

- Matrices are invariant under multiplication by the identity matrix.

$$AI = A$$

$$IA = A$$

Helpful matrix inversion properties

$$(A^{-1})^{-1} = A$$

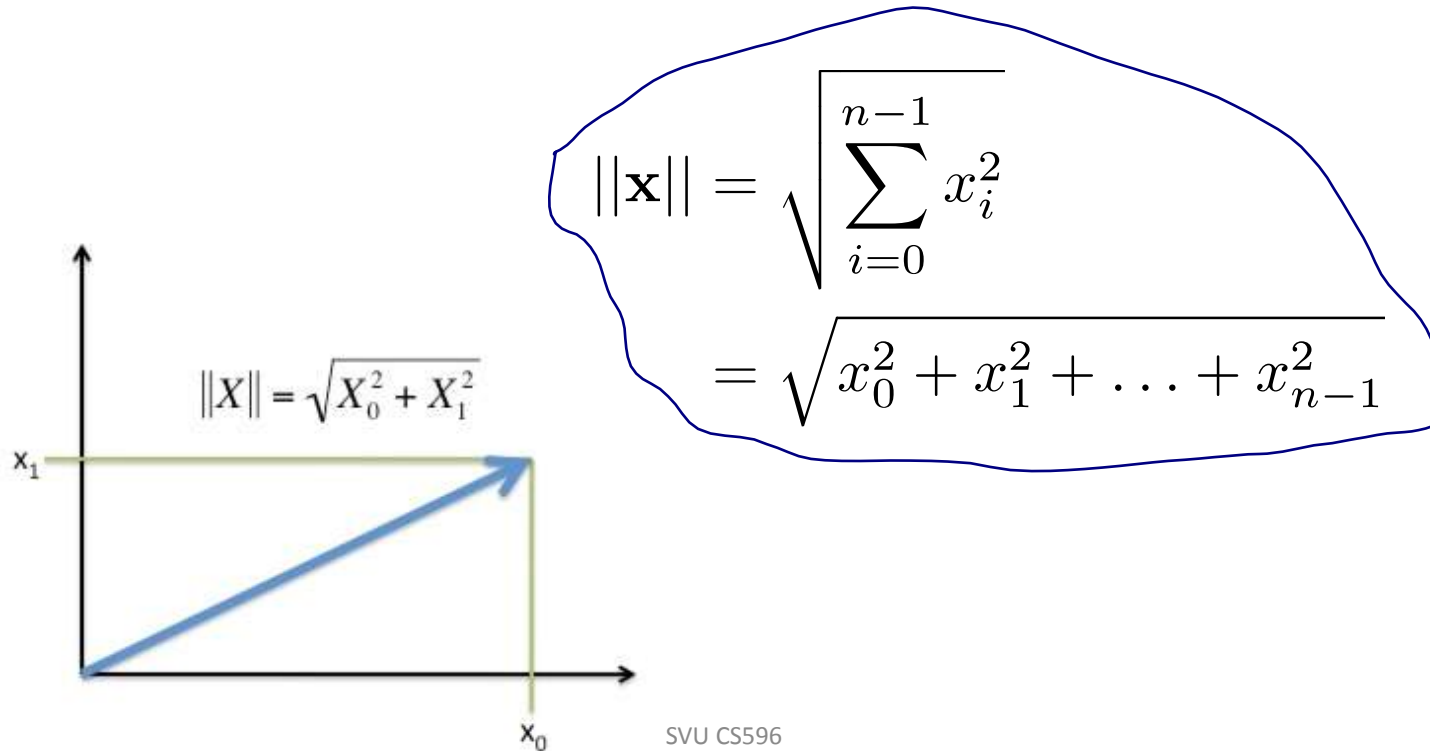
$$(kA)^{-1} = k^{-1}A^{-1}$$

$$(A^T)^{-1} = (A^{-1})^T$$

$$(AB)^{-1} = B^{-1}A^{-1}$$

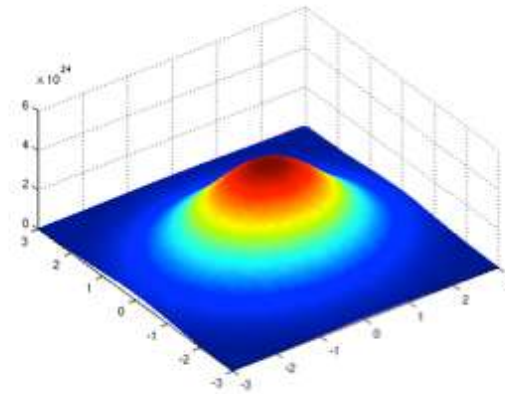
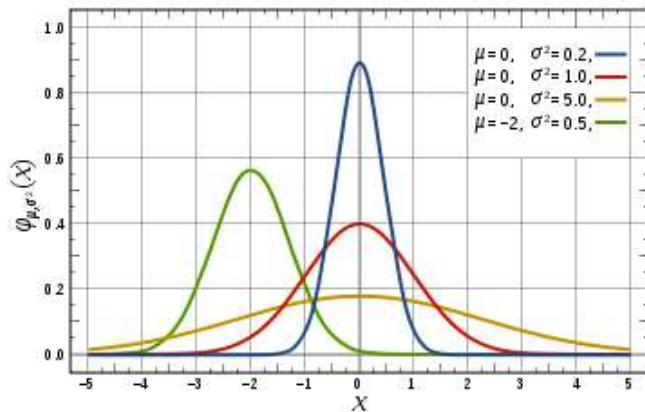
Norm

- The **norm** of a vector, \mathbf{x} , represents the Euclidean length of a vector.



Why do we need so much math?

- Probability Density Functions allow the evaluation of how likely a data point is under a model.
 - Want to identify good PDFs. (calculus)
 - Want to evaluate against a known PDF. (algebra)



Gaussian Distributions

- 1-D Gaussian Distributions

$$N(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left\{ -\frac{1}{2\sigma^2} (x - \mu)^2 \right\}$$

\uparrow mean
 \uparrow variance.
 \uparrow mean

mean shifts the graph
 variance expands the graph.

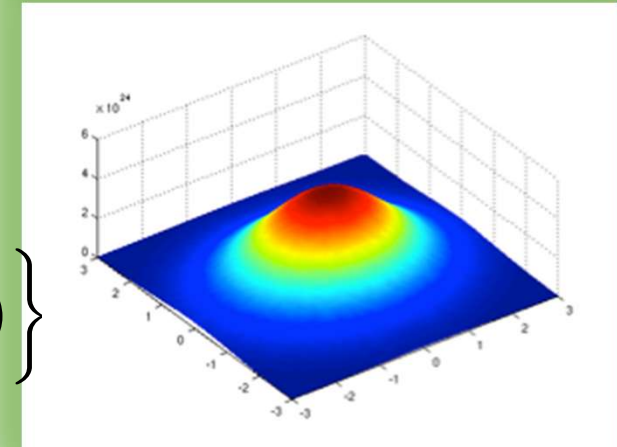
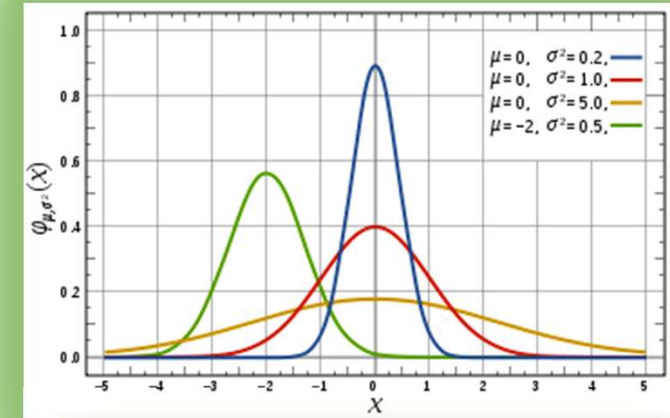
- D-dim Gaussian Distributions

$$N(x; \mu, \Sigma) = \frac{1}{(2\pi)^{D/2} |\Sigma|^{1/2}} \exp \left\{ -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right\}$$

\uparrow
inverse of sigma

5/22/2015

SVU CS596



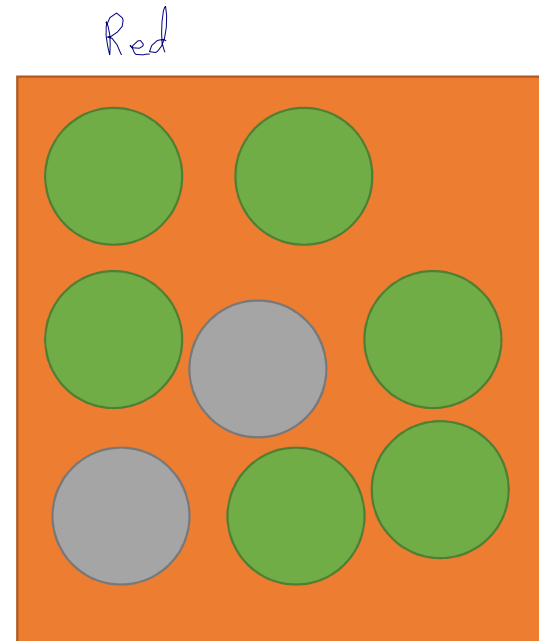
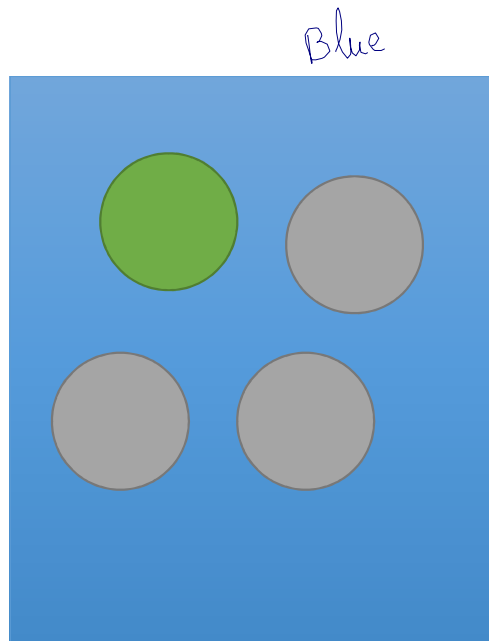
76

Examples

- 1D Gaussian: `plotGaussian.py`
- 2D Gaussian: `Gaussian.py`

Boxes and Balls

- 2 Boxes, one red and one blue.
- Each contain colored balls.

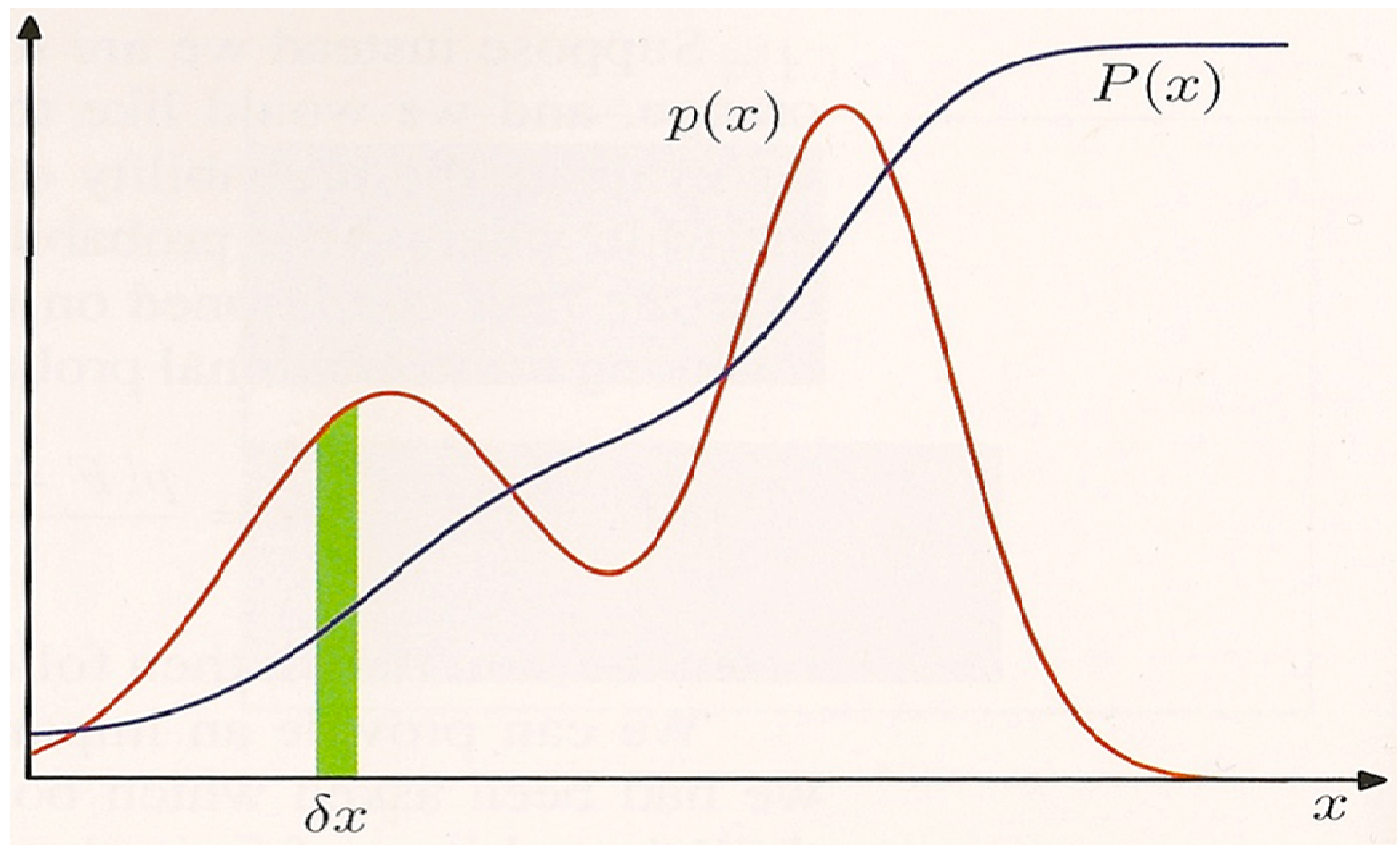


Continuous Probabilities

- So far, X has been **discrete** where it can take one of M values. *defined event by event*
- What if X is continuous?
- Now $p(x)$ is a continuous **probability density function**.
- The probability that x will lie in an interval (a,b) is:

$$p(x \in (a, b)) = \int_b^a p(x) dx$$

Continuous probability example



Properties of probability density functions

$$~~p(x) \geq 1~~ \quad p(x) > 1$$

$$\int_{-\infty}^{\infty} p(x) dx = 1$$

Sum Rule

$$p(x) = \int p(x, y) dy$$

Product Rule

$$\underline{p(x, y) = p(y|x)p(x)}$$

Expected Values

- Given a random variable, with a distribution $p(X)$, what is the expected value of X ?

x is discrete, e.g. roll a dice

$$\mathbb{E}[x] = \sum_x p(x)x$$

x is continuous.

$$\mathbb{E}[x] = \int p(x)x dx$$

density func

Multinomial Distribution

- If a variable, x , can take 1-of- K states, we represent the distribution of this variable as a **multinomial** distribution.
- The probability of x being in state k is μ_k


$$\sum_{k=1}^K \mu_k = 1 \qquad p(x; \boldsymbol{\mu}) = \prod_{k=1}^K \mu_k^{x_k}$$

Expected Value of a Multinomial

- The expected value is the mean values.

$$\mathbb{E}[\mathbf{x}; \boldsymbol{\mu}] = \sum_x p(x; \boldsymbol{\mu})x = (\mu_0, \mu_1, \dots, \mu_{K-1})^T$$

How machine learning uses statistical modeling

- Expectation
 - The expected value of a function is the **hypothesis** 
- Variance
 - The variance is the **confidence** in that hypothesis

	Orange	Green.
Value	1	2.
Prob	$\frac{7}{12}$	$\frac{5}{12}$

$$E = 1 \times \frac{7}{12} + 2 \times \frac{5}{12} = \frac{17}{12}$$

$$\text{Var} = \frac{7}{12} \left(1 - \frac{17}{12}\right)^2 + \frac{5}{12} \left(2 - \frac{17}{12}\right)^2$$

$$\text{std} = \sqrt{\text{var}}$$

Variance

- The **variance** of a random variable describes how much variability around the expected value there is.
- Calculated as the expected squared error.

$$\text{var}[f] = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2]$$

$$\text{var}[f] = \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2$$

Covariance

- The covariance of two random variables expresses how they vary together. (therefore, if they have no common, their covariance is \emptyset)

$$\begin{aligned}\text{cov}[x, y] &= \mathbb{E}_{x,y}[(x - \mathbb{E}(x))(y - \mathbb{E}[y])] \\ &= \mathbb{E}_{x,y}[xy] - \mathbb{E}[x]\mathbb{E}[y]\end{aligned}$$

- If two variables are **independent**, their covariance equals zero.

Bayes Theorem

Boxes and Balls

- Given some information about B and L , we want to ask questions about the likelihood of different events.
- What is the probability of selecting an apple?
- If I chose an orange ball, what is the probability that I chose from the blue box?

Some basics

- The **probability (or likelihood)** of an event is the fraction of times that the event occurs out of n trials, as n approaches infinity.
- Probabilities lie in the range $[0,1]$
- **Mutually exclusive** events are events that cannot simultaneously occur.
 - The sum of the likelihoods of all mutually exclusive events must equal 1.
- If two events are **independent** then,

$$p(X, Y) = p(X)p(Y)$$

conditioning prob $\rightarrow p(X|Y) = p(X) \leftarrow$ bec X is independent of Y .

\uparrow
knowing Y already happened,
the probability of X .

Joint Probability – P(X,Y)

- A Joint Probability function defines the likelihood of two (or more) events occurring.

	Orange	Green	
Blue box	1	3	4
Red box	6	2	8
	7	5	12

- Let n_{ij} be the number of times event i and event j simultaneously occur.

$$p(X = \overset{\text{balls}}{x_i}, Y = \overset{\text{boxes.}}{y_j}) = \frac{n_{ij}}{N (= \sum_i \sum_j n_{ij})}.$$

$$P(X = \text{Orange } \textcircled{1}, Y = \text{blue}) = \frac{n_{\text{orang, blue.}}}{n_{\text{orang, blue.}} + n_{\text{green, blue.}}} = \frac{1}{4}.$$

Generalizing the Joint Probability

	n_{ij}		$r_i = \sum_j n_{ij}$
	$c_j = \sum_i n_{ij}$		$\sum_i \sum_j n_{ij} = N$

	Orange ball	Green ball
Blue box	1	3
Red box	6	2

Marginalization

- Consider the probability of X irrespective of Y.

$$p(X = x_j) = \frac{c_j}{N}$$

- The number of instances in column j is the sum of instances in each cell

$$c_j = \sum_{i=1}^L n_{ij}$$

- Therefore, we can **marginalize** or “sum over” Y:

$$p(X = x_j) = \sum_{i=1}^L p(X = x_j, Y = y_i)$$

5/22/2015

derive:
flow

$$\left(\begin{aligned} p(X=x_j) &= \frac{c_j}{N} \\ &= \frac{\sum_{i=1}^L n_{ij}}{N} = \sum_{i=1}^L p(x_i, y_j) \end{aligned} \right)$$

SVU CS596

93

Conditional Probability

- Consider only instances where $X = x_j$.
- The fraction of these instances where $Y = y_i$ is the conditional probability
 - “The probability of y given x ”

$$p(Y = y_i | X = x_j) = \frac{n_{ij}}{c_j}$$

$$p(X = x_j | Y = y_i) = \frac{n_{ij}}{r_i}$$

Relating the Joint, Conditional and Marginal

$$\begin{aligned} p(X = x_i, Y = y_j) &= \frac{n_{ij}}{N} = \frac{n_{ij}}{c_i} \cdot \frac{c_i}{N} \\ &= p(Y = y_j | X = x_i) p(X = x_i) \end{aligned}$$

$$\Rightarrow \begin{cases} P(x, y) = P(y|x) P(x) \\ \text{or } P(y, x) = P(x|y) P(y) \end{cases}$$

Sum and Product Rules

- In general, we'll refer to a distribution over a random variable as $p(X)$ and a distribution evaluated at a particular value as $p(x)$.

Sum Rule

$$p(X) = \sum_Y p(X, Y)$$

Product Rule

$$p(X, Y) = p(Y|X)p(X)$$

this is used to derive Bayes' theory.

Bayes Rule

likelihood: the likelihood of X happens given Y

Posterior

Prior: info we have
b4 observation

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

derivation.

$$p(Y|X) \cdot p(X) = p(X|Y) p(Y)$$
$$P(Y, X) = P(X, Y)$$

Interpretation of Bayes Rule

The diagram shows the Bayes' Rule formula with three components highlighted in blue boxes: 'Posterior' for $p(Y|X)$, 'Likelihood' for $p(X|Y)$, and 'Prior' for $p(Y)$. The denominator $p(X)$ is not highlighted.

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

- **Prior:** Information we have before observation.
- **Posterior:** The distribution of Y after observing X
- **Likelihood:** The likelihood of observing X given Y

Boxes and Balls with Bayes Rule

- Assuming I'm inherently more likely to select the red box (66.6%) than the blue box (33.3%).
- If I selected an orange ball, what is the likelihood that I selected the red box?
 - The blue box?

Boxes and Balls

$$p(B = r|L = o) = \frac{p(L = o|B = r)p(B = r)}{p(L = o)}$$

$$= \frac{\frac{6}{8} \frac{2}{3}}{\frac{7}{12}} = \frac{6}{7}$$

$$p(B = b|L = o) = \frac{p(L = o|B = b)p(B = b)}{p(L = o)}$$

$$= \frac{\frac{1}{4} \frac{1}{3}}{\frac{7}{12}} = \frac{1}{7}$$

Naïve Bayes Classification

- This is a simple case of a simple classification approach.
- Here the Box is the class, and the colored ball is a feature, or the observation.
- We can extend this Bayesian classification approach to incorporate more **independent** features.

Naïve Bayes Classification

- Some theory first.

$$c^* = \operatorname{argmax}_c p(c|x_1, x_2, \dots, x_n)$$

$$c^* = \operatorname{argmax}_c \frac{p(x_1, x_2, \dots, x_n|c)p(c)}{p(x_1, x_2, \dots, x_n)}$$

$$p(x_1, x_2, \dots, x_n|c) = p(x_1|c)p(x_2|c) \cdots p(x_n|c)$$

Naïve Bayes Classification

- Assuming independent features simplifies the math.

$$c^* = \operatorname{argmax}_c \frac{p(x_1|c)p(x_2|c) \cdots p(x_n|c)p(c)}{p(x_1, x_2, \dots, x_n)}$$

$$c^* = \operatorname{argmax}_c p(x_1|c)p(x_2|c) \cdots p(x_n|c)p(c)$$

Naïve Bayes Example Data

HOT	LIGHT	SOFT	RED
COLD	HEAVY	SOFT	RED
HOT	HEAVY	FIRM	RED
HOT	LIGHT	FIRM	RED
COLD	LIGHT	SOFT	BLUE
COLD	HEAVY	FIRM	BLUE
HOT	HEAVY	FIRM	BLUE
HOT	LIGHT	FIRM	BLUE
HOT	HEAVY	FIRM	?????

$$c^* = \operatorname{argmax}_c p(x_1|c)p(x_2|c) \cdots p(x_n|c)p(c)$$

Naïve Bayes Example Data

HOT	LIGHT	SOFT	RED
COLD	HEAVY	SOFT	RED
HOT	HEAVY	FIRM	RED
HOT	LIGHT	FIRM	RED
COLD	LIGHT	SOFT	BLUE
COLD	HEAVY	FIRM	BLUE
HOT	HEAVY	FIRM	BLUE
HOT	LIGHT	FIRM	BLUE
HOT	HEAVY	FIRM	?????

Prior: $p(c = red) = 0.5$
 $p(c = blue) = 0.5$

Naïve Bayes Example Data

HOT	LIGHT	SOFT	RED
COLD	HEAVY	SOFT	RED
HOT	HEAVY	FIRM	RED
HOT	LIGHT	FIRM	RED
COLD	LIGHT	SOFT	BLUE
COLD	HEAVY	SOFT	BLUE
HOT	HEAVY	FIRM	BLUE
HOT	LIGHT	FIRM	BLUE
HOT	HEAVY	FIRM	????

$$\begin{aligned} p(hot|c = red) &= 0.75 & p(heavy|c = red) &= 0.5 & p(firm|c = red) &= 0.5 \\ p(hot|c = blue) &= 0.5 & p(heavy|c = blue) &= 0.5 & p(firm|c = blue) &= 0.5 \end{aligned}$$

Naïve Bayes Example Data

HOT	LIGHT	SOFT	RED
COLD	HEAVY	SOFT	RED
HOT	HEAVY	FIRM	RED
HOT	LIGHT	FIRM	RED
COLD	LIGHT	SOFT	BLUE
COLD	HEAVY	SOFT	BLUE
HOT	HEAVY	FIRM	BLUE
HOT	LIGHT	FIRM	BLUE
HOT	HEAVY	FIRM	????

$$p(hot|c = red)p(heavy|c = red)p(firm|c = red)p(c = red) = 0.75 * 0.5 * 0.5 * 0.5 \\ = 0.09375$$

$$p(hot|c = blue)p(heavy|c = blue)p(firm|c = blue)p(c = blue) = 0.5 * 0.5 * 0.5 * 0.5 \\ = 0.0625$$