Scientific Computing with Python

Eric Franzosa, Ph.D. (franzosa@hsph.harvard.edu)
2018-10-16

Outline

- Announcements
- Final project details
- The Python scientific stack
- `numpy` and its `ndarray`
- `pandas` and its `DataFrame`
- Brief overview of `scipy`, `statsmodels`, and `scikit-learn`
- Time-permitting: Use cases from my research

The Python *Scientific Stack*

Originally:

- `numpy` for its efficient array data structure
- `scipy` for numerical analysis methods
- `matplotlib` for 2D plotting

Today we also include:

- `pandas` a powerful data frame object
- `scikit-learn` for clustering and classification (machine learning)
- `statsmodels` for statistical modeling

`numpy`

- `numpy` is an important module *outside* of the Python standard library
- It is bundled with Anaconda 3
- The heart of scientific computing in Python

In [104]:

```
# numpy is imported under the shortcut <np>
import numpy as np
```

the `numpy.ndarray`

- At the core of numpy is the `numpy.ndarray`
- A class representing an *n*-dimensional array
- Vectors of numbers are 1-dimensional arr ays
    - *We've represented these as lists*
- Matrices of numbers are 2-dimensional arr ays
    - *We've represented these as lists of lists*
- `numpy.ndarray`s can have arbitrarily many dimensions
    - *though 1 and 2 are most common*

In [ ]:

```
help( np.ndarray )
```

making an `numpy.ndarray`

- We typically make `numpy.ndarrays` with the convenience function `numpy.array`
- For this reason, we often shorthand `numpy.ndarray` to just "array"
- We can turn a Python `list` into an array, for example:

In [106]:
```
np.array( [1,2,3,5,7] )
```

Out[106]:
```
array([1, 2, 3, 5, 7])
```

In [107]:
```
# let's put that array in the variable <a> so we can work with it
a = np.array( [1,2,3,5,7] )
```

In [108]:
```
# we can index into an array like a list
a[0]
```

Out[108]:
```
1
```

In [109]:
```
# we can slice from an array like a list as well
a[0:3]
```

Out[109]:
```
array([1, 2, 3])
```

In [110]:
```
# we can similarly define a 2d array
b = np.array( [[1,2,3],[4,5,6],[7,8,9]] )
```

In [111]:
```
# the "dimensions" of an array are stored in an attribute, <shape>
b.shape
```

Out[111]:
```
(3, 3)
```

In [112]:
```
# the length of <shape> is the dimensionality of the array
len( b.shape )
```

Out[112]:
```
2
```

In [113]:
```
# slicing/indexing a 2d array is easier than with a list of lists
b
```

Out[113]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [114]:
```
# indexing directly into the 2d array gives a 1d array
b[0]
```

Out[114]:
```
array([1, 2, 3])
```

In [115]:
```
# unlike lists of lists, we can index other dimensions using the <:> operato
r
b[:,0]
```

Out[115]:
```
array([1, 4, 7])
```

In [116]:
```
# we can slice out a chunk of the array easily
b[1:,1:]
```

Out[116]:
```
array([[5, 6],
       [8, 9]])
```

Mathematical operations on arrays

- Behave *very differently* from operations on `lists`
- More "mathy" (but not exactly the same as math!)

In [117]:

```
# duplication
2 * [1,2,3]
```

Out[117]:

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

In [118]:

```
# multiplication by a constant
2 * np.array( [1,2,3] )
```

Out[118]:

```
array([2, 4, 6])
```

In [119]:
```
# concatenation
[1,2,3] + [1,2,3]
```
Out[119]:
```
[1, 2, 3, 1, 2, 3]
```
In [120]:
```
# element-wise addition
np.array( [1,2,3] ) + np.array( [1,2,3] )
```
Out[120]:
```
array([2, 4, 6])
```

Some aspects of array math have no list equivalents

In [121]:
```
# broadcasting
1 + np.array( [1,2,3] )
```

Out[121]:
```
array([2, 3, 4])
```

In [122]:
```
# will not work
1 + [1,2,3]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-122-a5ced438c221> in <module>()
      1 # will not work
----> 2 1 + [1,2,3]

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

In [ ]:

```
# element-wise product
np.array( [1,2,3] ) * np.array( [1,2,3] )
```

In [ ]:

```
# will not work
[1,2,3] * [1,2,3]
```

- Element-wise multiplication works on pairs of arr ays with the same shape

In [ ]:
```
A = np.array( [[1,2],[3,4]] )
A
```

In [ ]:
```
B = np.array( [[0,1],[1,0]] )
B
```

In [ ]:
```
A * B
```

In [ ]:
```
# use <np.matmul> for proper matrix multiplication of 2d arrays
np.matmul( A, B )
```

Advanced indexing

In [ ]:
```
a = 2 * np.array( range( 9 ) )
a
```

In [ ]:
```
# index by a list of positions
a[[1,3,5]]
```

In [ ]:
```
# index with a yes/no (True/False) call for each position
a % 3 == 0
```

In [ ]:
```
a[a % 3 == 0]
```

Vectorized functions

- Arrays are designed to be used with *vectorized* functions
- Vectorized functions avoid explicit loops over arrays, which are slow (relatively speaking)
- `numpy` contains many vectorized versions of common functions

In [ ]:
```
b = np.array( [[1,2,3],[4,5,6],[7,8,9]] )
b
```

In [ ]:
```
# the sum of all 2d array elements
np.sum( b )
```

In [ ]:
```
# the sum over the first axis (note base-0 counting)
np.sum( b, axis=0 )
```

In [ ]:
```
# the sum over the second axis
np.sum( b, axis=1 )
```

Arrays and vectorized computation are fast

In [ ]:

```
# let's make a big list and array equivalent
import random
x = [random.random( ) for k in range( 100000 )]
y = np.array( x )
```

In [ ]:

```
%timeit -n100 np.sum( y )
```

In [ ]:

```
%timeit -n100 sum( x )
```

In [ ]:

```
# a manual loop is even slower
def my_sum( numbers ):
    ret = 0
    for n in numbers:
        ret += n
    return ret
```

In [ ]:

```
%timeit -n100 my_sum( x )
```

Other ways to make arrays

In [ ]:

```
# an array of ones of a specified shape
np.ones( [2, 3] )
```

In [ ]:

```
# an array of zeroes of a specified shape (note the spelling)
np.zeros( [2, 3] )
```

In [ ]:

```
# range equivalent
np.arange( 0, 2, 0.25 )
```

In [ ]:
```
# load from a file (by default splits rows on any whitespace)
a = np.genfromtxt( "numbers-10rows.tsv"  )
a
```

In [ ]:
```
# replicating elements of hw4
column = 3
print( "MEAN=", np.mean( a[:, column] ), "MEDIAN=", np.median( a[:, column]
) )
```

Element types

- Arrays try to force their elements to be floating-point numbers.
- We can tell arrays to coerce data to another type.
- Unlike `lists`, all array elements have to have the save type.
  - *This is part of where the efficiency of arrays comes from.*

In [ ]:
```
# casting to integers
np.genfromtxt( "numbers-10rows.tsv" , dtype=int )[0:3]
```

I use 1d arrays a lot for working with biological sequences

- Say I have a gene that is 50 nucleotides (nts) long...
- I map RNA-seq reads of length 15 nts starting at positions 12, 15, and 21...
- What is the coverage at position 17?

In [ ]:
```
gene = np.zeros( 50 )
read_starts = [12, 15, 17]
read_len = 15
for start in read_starts:
    start = start - 1
    gene[start:start+read_len] += 1
```

In [ ]:
```
gene
```

In [ ]:
```
gene[17-1]
```

What is a data frame?

- A data frame is special kind of 2d array
- Each row represents one "sample" or "observation"
    - *These may be named, but it's not requires*
- Each column represents a particular type of measurement
    - *Each column must have a unique name*
    - *The data in a single column are all of the same type*
- Very common (and important) in all sorts of statistical modeling

The `pandas` DataFrame

- The `pandas` module implements a powerful data fr ame class

In [ ]:

```
# pandas is typically imported as pd
import pandas as pd
```

In [ ]:

```
# we can build a DataFrame with a dictionary of lists (or arrays) of the sam
e length
data = {
    "Name": ["Alice", "Bob", "Carol"],
    "Height": [1.45, 1.83, 1.34],
    "Age": [23, 45, 91],
}
```

In [ ]:

```
# data frames are often abbreviated df
pd.DataFrame( data )
```

- It is much more common to load these sort of data from a file

In [ ]:

```
# note that the file doesn't have to be a csv, despite the name of the metho
d
df = pd.read_csv( "iris_renamed.tsv" , sep="\t" )
```

In [ ]:

```
df.head( )
```

Indexing columns

In [ ]:
```
# We can index particular columns using their names like dictionary keys
df["label"].head( )
```

In [ ]:
```
# the read_csv method makes smart choices about data types
df["petal_width"].head( )
```

In [ ]:

```
# can also access columns with "namespace"-style naming
df.petal_width.head( )
```

Indexing rows

In [ ]:
```
# rows are indexed with the .iloc attribute
df.iloc[0]
```

In [ ]:
```
# slicing works too
df.iloc[3:8]
```

Advanced row indexing

In [ ]:
```
# Like arrays, we can slice rows using Boolean vectors
df[df.sepal_width > 7.5]
```

In [ ]:
```
# pandas also includes convenience selection methods
df.nlargest( 5, "sepal_length" )
```

Data Exploration

- Pandas includes a lot of useful functions for data exploration

In [ ]:
```
# descriptive statistics for numerical columns
df.describe( )
```

In [ ]:

```
# same idea, but only for rows of a certain label
df[df.label == "Iris-setosa"].describe( )
```

In [ ]:

```
# regroup a data frame by a categorical feature
groups = df.groupby( "label" )
```

In [ ]:

```
# aggregate groups by some function
groups.agg( "mean" )
```

Going further with arrays and data frames

- Most scientific computing in Python can be done with basic data types (`lists`, `dicts`)
- Working with `numpy` arrays and/or `pandas` data frames is often easier and generally faster
- The `numpy.ndarray` and `pandas.DataFrame` are individually very powerful and contain many useful methods
- Google/consult the docs as needed

The `scipy` module

- Contains a wide variety of functions for scientific data analysis
- Examples: optimization, clustering, signal and image processing, and statistical testing
- Increasingly being broken apart into specialized "scientific kits" (scikits)

In [ ]:
```
from scipy.stats import mannwhitneyu, spearmanr
```

In [ ]:
```
pw_s = df[df.label == "Iris-setosa"].petal_width
sw_s = df[df.label == "Iris-setosa"].sepal_width
pw_v = df[df.label == "Iris-virginica"].petal_width
```

In [ ]:
```
mannwhitneyu( pw_s, pw_v )
```

In [ ]:
```
spearmanr( pw_s, sw_s )
```

The `statsmodels` module

- Regression analysis in Python using R-lik e syntax

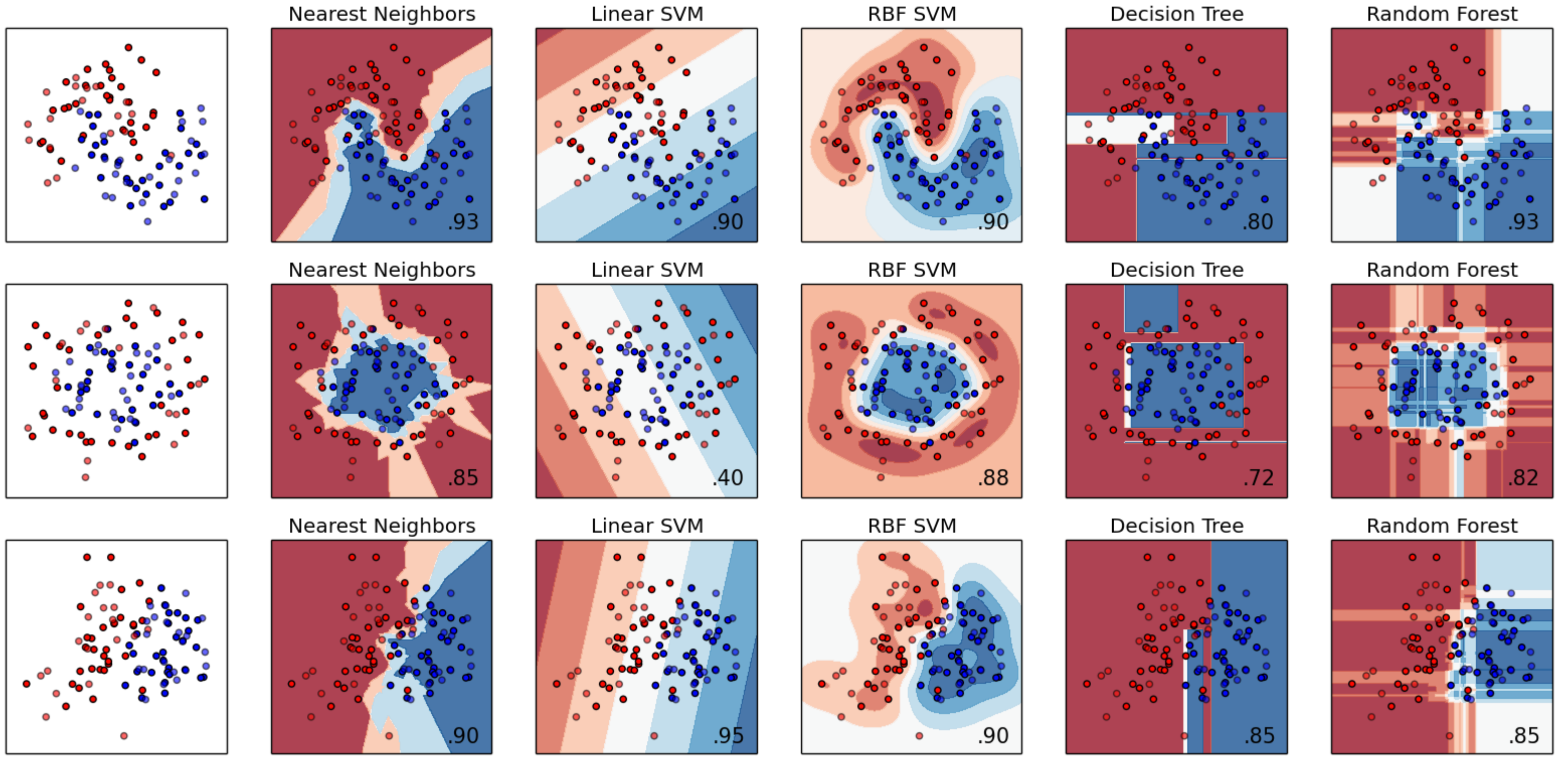
In [ ]:
```
import statsmodels.formula.api as smf
```

In [ ]:
```
results = smf.ols( "petal_width ~ sepal_width + C(label)" , data=df ).fit( )
```

In [ ]:
```
results.summary( )
```

The `scikit-learn` module

- Machine learning in Python
- Great website: [http://scikit-learn.org/stable/index.html (http://scikit-learn.org/stable/index.html)](http://scikit-learn.org/stable/index.html)
- Really nice for clustering and classification


Nearest Neighbors
Linear SVM
RBF SVM
Decision Tree
Random Forest
.93
.90
.90
.80
.93
Nearest Neighbors
Linear SVM
RBF SVM
Decision Tree
Random Forest
.85
.40
.88
.72
.82
Nearest Neighbors
Linear SVM
RBF SVM
Decision Tree
Random Forest
.90
.95
.90
.85
.85

Final Project Interactions

- Plots on the previous slide were generated with `matplotlib`
  - *Python's main plotting engine*
- You will learn about `matplotlib` for the final project
- You do not **need** to use arrays or data frames for the final project
  - *But you can if desired*

Outline

- Announcements
- Final project details
- The Python scientific stack
- `numpy` and its `ndarray`
- `pandas` and its `DataFrame`
- Brief overview of `scipy`, `statsmodels`, and `scikit-learn`
- **Time-permitting: Use cases from my research**