Python and Data Science

Chris Jermaine

Rice University

Python

- Old language, first appeared in 1991
  - But updated often over the years
- Important characteristics
  - Interpreted
  - Dynamically-typed
  - High level
  - Multi-paradigm (imperative, functional, OO)
  - Generally compact, readable, easy-to-use
- Boom in popularity last five years
  - Now the first PL learned in many CS departments

Python: Why So Popular for Data Science?

- Dynamic typing/interpreted
  - Type a command, get a result
  - No need for compile/execute/debug cycle
- Quite high-level: easy for non-CS people to pick up
  - Statisticians, mathematicians, physicists...
- More of a general-purpose PL than R
  - More reasonable target for larger applications
  - More reasonable as API for platforms such as Spark
- Can be used as lightweight wrapper on efficient numerical codes
  - Unlike Java, for example

Python Basics

- Since Python is interpreted, can just fire up Python shell
  - Then start typing

- A first Python program

```
def Factorial (n):
     if n == 1 or n == 0:
          return 1
     else:
          return n * Factorial (n - 1)

Factorial (12)
```

Python Basics Continued

- Spacing and indentation
  - Indentation important
  - No begin/end nor
  - Indentation signals code block

- Variables
  - No declaration
  - All type checking dynamic
  - Just use

Python Basics Continued

- Dictionaries
  - Standard container type is dictionary/map
  - Example: `wordsInDoc = {}` creates empty dictionary
- Adding Data
  - Add data by saying `wordsInDoc[23] = 16`
  - Now can write something like `if wordsInDoc[23] == 16:` ...
  - What if `wordsInDoc[23]` is not there? Will crash
  - Protect with `if wordsInDoc.get (23, 0)..a.` returns 0 if key 23 not defined

Encapsulation

- Functions/Procedures
  - Defined using `def myFunc (arg1, arg2):`
  - Make sure to indent!
  - Procedure: no return statement
  - Function: return statement

- Remember:
  - No marker to end func/proc
  - It ends when you stop indenting

Loops

- Several common forms

- Looping through a range of values
  - Of form for `var in range (0, 50)`
  - Loops for `var` in `{0, 1, ..., 49}`

- Looping through data structures
  - Example: `for var in dataStruct`
  - loops through each entry in `dataStruct`
  - `dataStruct` can be an array, or a dictionary
  - If array, you loop through the entries
  - If dictionary, you loop through the keys

Loops Continued

- An example

```
a = {}
a[1] = 'this'
a[2] = 'that'
a[3] = 'other'
for b in a:
        a[b]

'this'
'that'
'other'
```

NumPy

- NumPy is a Python package
- Most important one for data science!
  - Can use it to do super-fast math, statistics
  - Most basic type is NumPy array
  - Used to store vectors, matrices, tensors
- You will get some reasonable experience with NumPy
- Load with `import numpy as np`

NumPy Arrays: What Are They?

- Multi-dimensional array data structure
- And associated API
- Widely used for data intensive programming...
  - Linear algebra
  - Data science
  - ML

NumPy Arrays: Your Best Friend In DS

- Writing control flow code in DS programming is BAD
  - ▷ Kind of like in SQL
- Python is interpreted
  - ▷ Time for each statement execution generally large
  - ▷ And in DS, you have a lot of data
  - ▷ So this code can take a long time:

```
for b in range(0, BIG):
    a[b] = b

sum = 0
for b in a:
    sum += a[b]
```

- Fewer statements executed, even if work same...
  - ▷ ...means better performance!

To Reduce Number of Statements...

- Use NumPy arrays where possible
- Goal: one line of Python to process entire array!
- Some guidelines:
  - Try to replace dictionaries with NumPy arrays
  - Try to replace loops with bulk array operations
  - Backed by efficient, low-level implementations
  - This is known as "vectorized" programming

Creating and Filling NumPy Arrays

- To create a 2 by 5 array, filled with 3.14

```
>>> np.full((2, 5), 3.14)

array([[ 3.14,  3.14,  3.14,  3.14,  3.14],
       [ 3.14,  3.14,  3.14,  3.14,  3.14]])
```

- To create a 2 by 5 array, filled with 0

```
>>> np.zeros((2, 5))

array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

More Complicated Creation Examples

- To create an array with odd numbers thru 10

```
>>> np.arange(1, 11, 2)

array([1, 3, 5, 7, 9])
```

- To “tile” an array

```
>>> np.tile (np.arange(1, 11, 2), (1, 2))

array([[1, 3, 5, 7, 9, 1, 3, 5, 7, 9]])

>>> np.tile (np.arange(1, 11, 2), (2, 1))

array([[1, 3, 5, 7, 9],
       [1, 3, 5, 7, 9]])
```

Accessing Subparts of Arrays

- First we create a 2-d array (matrix)

```
>>> a1 = np.arange(1, 6, 1)
>>> a2 = np.arange(2, 7, 1)
>>> a3 = np.arange(3, 8, 1)
>>> a = np.row_stack ((a1, a2, a3))
>>> a

array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

Accessing Subparts of Arrays (cont)

correction: "last two rows"

- Say we want first two rows:

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

array([[2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])

>>> a[1:]
array([[2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

- Why does this work?

- Gets rows 1, 2, 3, and so on

Accessing Subparts of Arrays (cont)

- Say we want the last row:

```
>>> a[2:3,]
array([[3, 4, 5, 6, 7]])

>>> a[2:3]
array([[3, 4, 5, 6, 7]])
```

- Note: still a 2-d array. Want a vector?

```
>>> a[2:3][0]

array([3, 4, 5, 6, 7])
```

Accessing Subparts of Arrays (cont)

- Now we want the second, third columns:

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

array([[2, 3],
       [3, 4],
       [4, 5]])

>>> a[:,np.array((1,2))]

array([[2, 3],
       [3, 4],
       [4, 5]])
```

- Works because `np.array((1,2))` is the array `[1, 2]`
- `a[:,np.array((1,2))]` gives you all rows, columns 1, 2

Aggregations Over Arrays

- In statistical/data analytics programming...
  - Tabulations: max, min, etc. over NumPy arrays are ubiquitous

- Key operation allowing this is sum

```
>>> a = np.arange(1, 6, 1)
>>> a

array([1, 2, 3, 4, 5])

>>> a.sum ()

15
```

Aggregations Over Arrays (cont)

- Can sum along dimension(s) of higher-d array

```
>>> a

array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])

>>> a.sum (0)

array([6, 9, 12, 15, 18])

>>> a.sum (1)

array([15, 20, 25])
```

Aggregations Over Arrays (cont)

- Can find the maximum the same way

```
>>> a
array([[10, 2,  3, 4, 5],
       [ 2, 3, 13, 5, 6],
       [ 3, 4,  5, 6, 7]])

>>> a.max ()

13

>>> a.max (0)

array([10, 4, 13, 6, 7])

>>> a.max (1)

array([10, 13, 7])
```

Aggregations Over Arrays (cont)

- Can find the position of the max as well

```
>>> a

array([[10, 2,  3, 4, 5],
       [ 2, 3, 13, 5, 6],
       [ 3, 4,  5, 6, 7]])

>>> a.argmax ()

7

>>> a.argmax (1)

array([0, 2, 4])
```

Now We Need a “Real Life” Set of Problems

- ...where we can apply some of these ideas

Latent Dirichlet Allocation (LDA)

- We will use data created by a statistical model called "LDA"
- LDA: stochastic model for generating a document corpus
- Most widely-used "topic model"
- A "topic" is a set of words that appear together with high prob
  - Intuitively: set of words that all have to do with the same subject

LDA Typically Used To Analyze Text

- Idea
  - If you can analyze a corpus...
  - And figure out a set of k topics...
  - As well as how prevalent each topic is in each document
  - You then know a lot about the corpus
  - Ex: can use this prevalence info to search the corpus
  - Two docs have similar topic compositions? Then they are similar!

Forward vs. Backward modeling

- Often, we want to "learn" an LDA model from an existing corpus
  - That is, you have a real data set
  - And you analyze the the data set
  - Goal: figure out how LDA model could have produced it
  - This is "backward" modeling
- But can also use it to generate a corpus
  - "Forward" modeling using LDA far less common
  - But we'll use the forward LDA process to generate our lab data

Dictionary Models

- LDA is a “Bag of Words” model
- Does not impose ordering on words in doc
- Uses a dictionary
  - ▷ Dictionary is a map from each of m unique words in corpus
  - ▷ To a number from $\{1...m\}$
- Example:
  - ▷ Dictionary might be: (0, bad) (1, I) (2, can’t) (3, stand) (4, COMP101), (5, to) (6, leave) (7, love) (8, beer) (9, humanities) (10, classes)

From Dictionary to Bag of Words

- Document is a vector x
  - ▷ $x[i]$ (ith entry in vector) is number of times dictionary word i appears in doc

0 1 2 3 4 5 6 7 8 9 10...

- Recall our dictionary is (0, bad) (1, I) (2, can't) (3, stand) (4, COMP101), (5, to) (6, leave) (7, love) (8, beer) (9, humanities) (10, classes)

- Then

  freq for dict

  0 1 2 3 4 5 6 7 8 9 10

  - ▷ Sentence "I can't stand bad beer" is $\langle 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0 \rangle$
  - ▷ Sentence "I love beer I can't love humanities classes" is $\langle 0, 2, 1, 0, 0, 0, 0, 2, 1, 1, 1 \rangle$

  0 1 2 3 4 5 6 7 8 9 10

without order you have smaller dictionary but lose semantics, but you get back using k-grams.

LDA Step One

This example is used for generating corpus, assuming you already have a model. "Forward"

- Generate a list of the k "topics"
  - Each topic is represented by a vector of probabilities
  - $wordsInTopic_t[w]$ is the probability that topic t would produce word w
  - $wordsInTopic_t$ is sampled from a Dirichlet (α) distribution

- Example, $k = 3$
  - $wordsInTopic_0 = \langle .2, .2, .2, .2, 0, 0, 0, 0, .2, 0, 0 \rangle$

    vector of probs so adds up to 1
  - $wordsInTopic_1 = \langle 0, .2, .2, .2, 0, 0, 0, 0, 0, .2, .2 \rangle$
  - $wordsInTopic_2 = \langle 0, .2, .2, 0, .2, 0, .2, .2, 0, 0, 0 \rangle$

LDA Step Two

- Generate the topic proportions for each document
  - Each topic "controls" production of some of the words in a doc
  - $topicsInDoc_d[t]$ is the probability that an arbitrary word in document d will be controlled by topic t
  - $topicsInDoc_d$ is sampled from a Dirichlet (β) distribution

LDA Step Three

- Generate the bag of words in each document
- $wordsInDoc_d[w]$ is the number of occurrences of word w in document d
- To get this vector, generate the words one-at-a-time
- For each word in d
  - Figure out the topic t that controls it:
  - Sample from a Multinomial $(topicsInDoc_d, 1)$ distribution
  - Generate the word w by sampling from a Multinomial $(wordsInTopic_t, 1)$ dist
  - Then increment $wordsInDoc_d[w]$

toss ball to different sized buckets topic, then toss ball into different sized buckets for word within topic

Example

- $topicsInDoc_0 = \langle .98, 0.01, 0.01 \rangle$

topic 0 is most important

- Generate first word:
  - We get $\langle 1, 0, 0 \rangle$ from a Multinomial $(topicsInDoc_0, 1)$ dist
  - So we generate the word using $wordsInTopic_0 = \langle .2, .2, .2, .2, 0, 0, 0, 0, .2, 0, 0 \rangle$
  - And we get $\langle 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, which is equivalent to “I”

- Generate the second word:
  - We get $\langle 1, 0, 0 \rangle$ from a Multinomial $(topicsInDoc_0, 1)$ dist
  - So we generate the word using $wordsInTopic_0 = \langle .2, .2, .2, .2, 0, 0, 0, 0, .2, 0, 0 \rangle$
  - And we get $\langle 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$, which is equivalent to “can’t”

- Generate the third word:
  - We get $\langle 1, 0, 0 \rangle$ from a Multinomial $(topicsInDoc_0, 1)$ dist
  - So we generate the word using $wordsInTopic_0 = \langle .2, .2, .2, .2, 0, 0, 0, 0, .2, 0, 0 \rangle$
  - And we get $\langle 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 \rangle$, which is equivalent to “stand”

generating is sequential but end result is bag of words so order is not important

And the Doc Generated...

- Recall the three words generated were:
  - ▷ $\langle 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$
  - ▷ $\langle 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$
  - ▷ $\langle 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 \rangle$
- Doc so far is $\langle 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 \rangle$
- Keep going and get $wordsInDoc_0 = \langle 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0 \rangle$
- Encodes "I can't stand bad beer"

~ I stand bad beer can't

OK, Back To Python!

- In a series of labs, we will look at some code that implements LDA
- Uses lots of NumPy functionality
  - `np.random.multinomial (numTrials, probVector, numRows)`
  - Take numRows samples from a Multinomial (probVector, numTrials) dist
  - Put in a matrix with numRows rows
  - `np.flatnonzero (array)`
  - Return array of indices of non-zero elements of array
  - `np.random.dirichlet (paramVector, numRows)`
  - Take numRows samples from a Dirichlet (paramVector) dist
  - `np.full (numEntries, val)`
  - Create a NumPy array with the spec'ed number of entries, all set to val

Questions?

First Activity: LDA

- Can you complete the activity?
  - `cmj4.web.rice.edu/LDADictionaryBased.html`

Problem: Bad Code!

- As we said: Don't write statistical/math Python code this way
- Vectorized is better!
- Can you complete the activity?
  - `cmj4.web.rice.edu/LDAArrays.html`
  - No dictionaries here! Just arrays

Computing Cross-Tabulations

- Now that we have an array-based LDA code...
- Let's practice doing cross-tabuations on it
- Can you complete the activity?
  - `http://cmj4.web.rice.edu/Subarrays.html`

Now We'll Implement Co-Occurrence Analysis

- Fundamental task in many statistical/data mining computations
- In text processing...
  - Given a document corpus
  - Want to count number of times $(word_1, word_2)$ occur in same doc in corpus
- Your task in next activity:
  - Build three implementations
  - Utilizing varying degrees of vectorization
  - We will time each, see which is faster

Implementation One

- Nested loops
- Loop through each doc...
  - For each doc, consider each (word, word) pair it contains
  - And increment the count
- Has advantage when $wordsInCorpus$ is sparse
  - Only $numDocs \times (numDistinctWordsPerDoc)^2$ execs of inner loop
- But not great in an interpreted language

Implementation Two

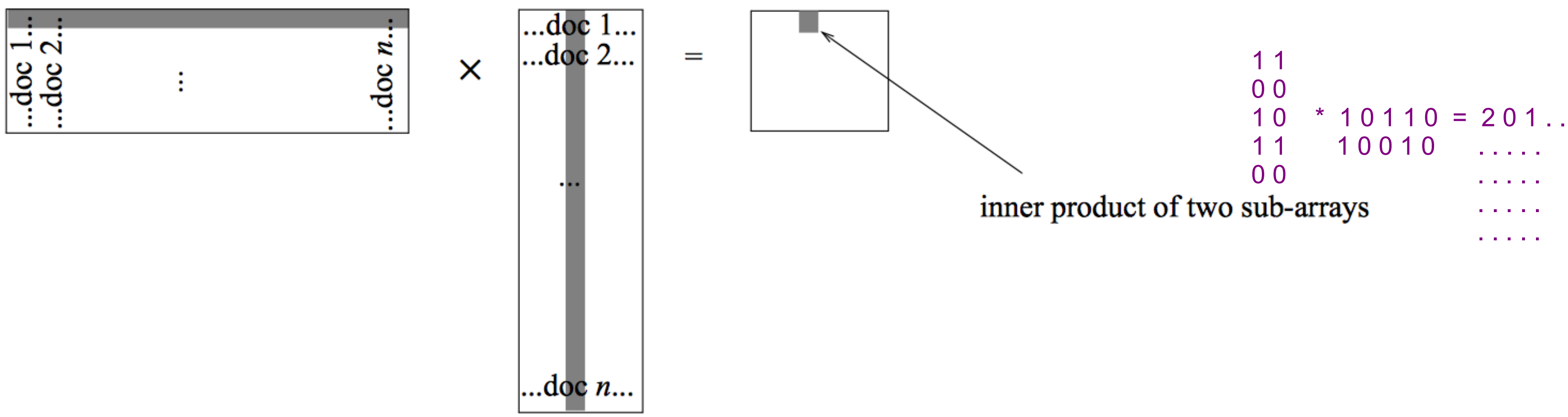
- Vector-based, with a loop over docs

- Given a 1-d array...
    - The outer product of array with itself creates a 2-d matrix
    - Where ith row is $array[i] \times array$
    - So if an array gives number of occurs of each word in a doc...
    - And we clip array so $\langle 0, 0, 3, 1, 0, 1 \ldots \rangle$ becomes $\langle 0, 0, 1, 1, 0, 1 \ldots \rangle$...
    - Then take outer product of array with itself...
    - Entry at pos (i, j) is number of co-occurs of dictionary words i, j in doc

outer product
<1, 4, 7>
*
<1, 4, 7>
1*1 1*4 1*7
1*4 4*4 7*4
etc

- Note:
    - `np.outer (arrayOne, arrayTwo)` is outer product of arrays
    - `np.clip (array, low, high)` clips all entries to max of high, min of low

Implementation Three



- Pure vector-based
- Note that after matrix multiply...
  - ▷ Entry at pos (i, j) is inner product of row i from LHS, col j from RHS
  - ▷ So if row i is number of occurs of word i in every doc
  - ▷ And if col j is number of occurs of word j in every doc
  - ▷ Entry at pos (i, j) is number of co-occurs of words i, j
  - ▷ Suggests a super-efficient algorithm

Implementation Three (cont)

- Some useful routines:
  - `np.transpose (array)` computes transpose of matrix in array
  - `np.dot (array1, array2)` computes dot product of 1-d arrays, matrix multiply of 2-d

These Three Implementations: The Next Activity

- Compare the three different implementations
  - http://cmj4.web.rice.edu/CoOccur.html

Questions?