CPSC 540: Machine Learning

Nando de Freitas, Eric Brochu and Matt Hoffman

1

Introduction

The purpose of machine learning is to enable machines to automatically understand data and, as a result, make useful predictions. By understanding, we mean that such machines will be able to derive abstractions from the data. We will refer to these abstractions as patterns or *features* . Some of the features will be embedded in structures (*e.g.* clusters, hierarchies, recursions, associations and other relations) to form complex features. Ideally, we would like to learn such embeddings and structures from data.

FEATURES

The quality of the features will be measured in terms of how well the machine can solve predictive tasks, such as inferring missing data, forecasting into the future, making decisions, detecting anomalies, creating summaries and classifying data instances. Another important desiderata is for the complex features to be transferable. That is, after learning structures in one domain, we would like to use those structures to accelerate learning in other domains. This is often referred to as transfer , multi-task or self-taught learning. As a toy illustration, someone living in an isolated game park in Southern Africa would most likely learn that "fangs" are features of "dangerous animals". Now imagine that person leaving Africa for the first time and encountering a bear in Vancouver. The learned features would most likely trigger the "dangerous animal" association.

PREDICTION

TRANSFER LEARNING

Many believe that we will eventually build learning machines that can match and outperform humans in complex tasks such as perception, motor control and probabilistic reasoning. The impact of this research program in science, industry and society will be very profound and lasting. For the time being, humans and other animals, still outperform machines in most of these tasks. For this reason, we often look at biological systems for inspiration to design better models and algorithms.

Machine learning is a fast growing field, which has already made substantial, and key, contributions to computational biology, search engines and recommender systems, credit scoring systems, machine vision systems, entertainment games and many other areas of human endeavor. We've only started to see the impact of machine learning in industry. A curious fact is that most humans seem oblivious to this. For example, personalized recommendation systems (such as Amazon, Zite and Netflix) learn users tastes and habits. Despite huge media coverage of these services, only a few blogs seem to have noted the significance of the fact that these are machines learning from humans. The title of one of these blogs is "*the quiet rise of machine learning*".

The rise of machine learning has coincided with great advances in computing and memory systems and with another widely discussed phenomenon: *big data* . Machine learning systems have become essential because our ability to acquire and store data has surpassed our ability to understand it. According to a study reported in the February 27th 2010 issue of *The Economist*, a typical individual in North America is bombarded with 34 Gigabytes of information at home per day. (In comparison, a typical two hour film can be compressed to 1 to 2 Gigabytes.) We are swamped with a deluge of data, which creates enormous research and business opportunities, but also new threats. It impacts medicine and health-care, our understanding of mind and intelligence, supply-chain management, privacy, public policy, and our capacity to improve energy usage and manage our natural resources.

BIG DATA

I should note that this rise has only been quiet in the general public. There is a huge demand at present in industry for skilled machine learning professionals.

2

Bringing Order to the Web: Information Retrieval with Python

This introductory chapter has two purposes. First, to familiarize you with Python. Second, to introduce you to the problem of ranking webpages — a key problem faced by search engines.

The chapter will quickly give you enough Python to get you through the first couple of weeks. *This is not nearly enough for the later exercises in this book*, and it is expected you will at the very least work through the relevant portions of the Python tutorial at `python.org` (which is excellent) and the NumPy tutorial at `scipy.org` (which is somewhat less excellent).

2.1 Importing

Python definitions (of, for example, classes, functions and submodules) are contained in *modules*, which are contained in files of the form `*.py`. To import modules into Python, use the `import` command. The following instruction

PYTHON MODULES

```
>>> import string
```

will import the string module, for string processing. You can then access definitions like this:

```
>>> string.lower('This is a Python string.')
this is a python string
```

You can also import definitions from modules directly, without importing the rest of the module using the `from` keyword:

```
>>> from string import lower
>>> lower("This, too, is a Python string.")
this, too, is a python string.
```

or even:

```
>>> from string import *
>>> swapcase("Strings can use 'double-quotes', too.")
sTRINGS CAN USE 'DOUBLE-QUOTES', TOO.
```

2.2 Containers

Python has a variety of containers built-in, and there are two important ones we will be importing from NumPy.

The built-in containers that are immediately relevant are lists, sets and dicts.

PYTHON LISTS

Lists are mutable, heterogeneous sequences. You can create lists by using `list()`, but there is also a special syntax: `[]`. In the Python interpreter, you can always see the value of a variable simply by typing the variable name.

```
>>> mylist = [100,200,300]
>>> mylist[1]
200
>>> a, b, c = mylist
>>> a
100
>>> b
200
```

SLICES

You can also get ranges and slices of a list using the `[`*start*`:`*end*`:`*step*`]` operator. Negative numbers count the array from the end.

```
>>> r = range(10)
>>> r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> r[:3]
[0, 1, 2]
>>> r[2:5]
[2, 3, 4]
>>> r[5:2:-1]
[5, 4, 3]
>>> r[1::2]
[1, 3, 5, 7, 9]
```

Now try the following commands, and variations until you are comfortable with how lists work.

```
>>> mylist.append('you can mix data types in a list')
>>> mylist
[100, 200, 300, 'you can mix data types in a list']
>>> mylist[2] /= 100
>>> mylist.sort()
>>> mylist
[3,100,200,'you can mix data types in a list']
>>> mylist.extend(['add', 'another', 'list'])
>>> mylist[1:3]
[100,200]
>>> mylist[:3]
[3,100,200]
>>> mylist[-1]
['list']
```

PYTHON TUPLES

Tuples are *immutable* sequences. They are very much like lists, except that their contents can't be changed after being created (actually, it's a little more complicated than that, but we won't get into that right now). This is necessary if you want to use a sequence as the key for a hash, and useful at other times. They can be created with the `()` syntax.

```
>>> mytuple = (12, 3, 4)
>>> mytuple[1]
3
>>> mytuple[1] = 10
--------------------
TypeError                                 Traceback (most
    recent call last)

/Users/eric/Dropbox/bookrepository/<ipython console> in <
    module>()

TypeError: 'tuple' object does not support item assignment
```

PYTHON SETS

Sets are not sequences: You cannot guarantee the order of items added to a set. They also have various set operations defined. All items in a set are unique.

```
>>> setA = set([1,2,3])
>>> setB = set()
>>> setB.add(3)
>>> setB.add('a string')
>>> setA & setB
set([3])
>>> setA | setB
set([1, 2, 3, 'a string'])
>>> setA - setB
set([1, 2])
```

PYTHON DICTS

Dicts are maps from keys to values. Like lists, they have a special syntax: {}. Like sets, the order is not guaranteed and keys must be unique.

```
>>> d = {'one': 1, 'two': 2}
>>> d['one']
>>> d['three'] = 3
>>> d['many'] = [1, 10, 100]
>>> 'three' in d
True
>>> 3 in d
False
>>> d.keys()
['many', 'three', 'two', 'one']
>>> d.values()
[[1, 2, 3], 3, 2, 1]
>>> d.items()
[('many', [1, 2, 3]), ('three', 3), ('two', 2), ('one', 1)]
```

Needless to say, there is much, much more you can do with these data structures, as well as more exotic ones like *defaultdicts* and *frozensets*.

2.3 Control flow

Python uses indentation for code blocks. You can use any number of tabs or spaces to set off a block, but your life is likely to be much easier if you consistently use 4 spaces.

For the remainder of the tutorial, the results aren't shown in the code snippets. Just remember, you can always see the value of a variable by typing it in the interpreter.

```
>>> x = 10
>>> if x > 5:
...     print 'x greater than 5'
>>> else:
...     print 'x <= 5'

>>> z = 0
>>> while z < 4:
...     print z
...     z += 1

>>> mylist = [3,4,5]
>>> for i in mylist:
...     print 'value = %d' % i
```

Note than unlike some languages, in Python `for` loops iterate over containers. While you *can* use indices:

```
>>> for ind in range(len(mylist)):
...    print 'value =', mylist[ind]
```

this is inefficient, unpythonic, and – not always, but *often* – means you are doing something wrong. You should probably be using `zip()`, `enumerate()` or the `itertools` module instead, especially if you aren't working with arrays or matrices.

LIST COMPREHENSIONS

List comprehensions are a fantastically useful way to create lists and apply map and filter operations. Like list declarations, they use the `[]` syntax, but they contain an expression and a `for` clause instead of values.

```
>>> from math import pow
>>> data = [10, 20, 30, 50]
>>> squared = [pow(x, 2) for x in data]
>>> squared
[100, 400, 900, 2500]
>>> filtered = [x for x in data if x > 25]
>>> filtered
[30, 50]
>>> bothed = [x**2 for x in data if x <= 20]
>>> bothed
[100, 400]
```

Common uses are to construct lists of lists and to initialize dicts.

```
>>> line =  something completely different
>>> pairs = [(x.upper(), len(x)) for x in line.split()]
>>> pairs
[('SOMETHING', 9), ('COMPLETELY', 10), ('DIFFERENT', 9)]
>>> eless = dict((x, x.replace('e','-')) for x in line.split
    ())
>>> eless['completely']
'compl-t-ly'
```

2.4 Files and functions

Python files end with a `*.py` extension. They can contain definitions, instructions, or both.

In Python, functions are defined with the `def` keyword followed by the arguments in parentheses. The `return` keyword passes the indicated object back the the caller. Functions can be declared from the Python session command line, but it is more common to put them in a file and import them. For example, if this is the content of `fdef.py`:

PYTHON FUNCTIONS

```
def foo(x, y):
    print x, '^', y
    z = x**y
    return z
```

you can call it from the session like this:

```
>>> from fdef import foo
>>> foo(10, 2)
10 ^ 2
100
>>> r = [foo(x, 2) for x in range(4)]
0 ^ 2
1 ^ 2
2 ^ 2
3 ^ 2
>>> r
[0, 1, 4, 9]
```

If you change the contents of a module, you must `reload` it. Note that this will *not* reload any modules that the file itself imports!

You can also put instructions in a file to run as a script. To run the script in the session, you can use `execfile()` (you can also use `import`, but in that case, you must `reload` to run it again). `myscript.py`:

```
line = 'and now for something completely different'
for word in line.split():
    es = sum([1 for x in word if x=='e'])
    print "'%s' e-count = %d" % (word, es)
```

can be run like this:

```
>>> execfile('myscript.py')
'and' e-count = 0
'now' e-count = 0
'for' e-count = 0
'something' e-count = 1
'completely' e-count = 2
'different' e-count = 2
```

2.5 NumPy Arrays

PYTHON ARRAYS

NumPy adds two very powerful containers: *arrays* and *matrices*. Matrices are more restrictive than arrays, but have some nice syntactical advantages that make them behave more like mathematical matrices. We'll be using arrays exclusively here – for details on matrices, see the NumPy website.

In all the remaining code examples, it is assumed you have already imported NumPy. Note that this adds NumPy overloads for Python built-ins such as `sum()` and `all()`.

```
>>> from numpy import *
```

Unlike lists, sets and dicts, array elements must all be the same type (which you can specify, if necessary with the `dtype=` parameter), and you cannot easily (or cheaply) resize them, though you can change individual elements.

```
>>> A = array([1.2, .2, 2.])
>>> sum(A)
3.4
>>> M = array([[1,2,3],[4,5,6],[7,8,9]], dtype=float)
>>> M
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  9.]])
>>> M.transpose()
array([[ 1.,  4.,  7.],
       [ 2.,  5.,  8.],
       [ 3.,  6.,  9.]])
>>> A.shape
(3,)
>>> M.shape
(3, 3)
```

The arithmetic operations (*, -, +, /, **) are defined as elementwise operators between arrays and sequences (including other arrays), or between arrays and scalars. For other linear algebra operations, NumPy has functions defined – `dot()` and `outer()` for inner and outer products, for example.

```
>>> A + 5.0
array([ 6.2,  5.2,  7. ])
>>> A + [1, 2, 3]
array([ 2.2,  2.2,  5. ])
>>> A**2
array([ 1.44,  0.04,  4.  ])
>>> dot(M, A)
array([  7.6,  17.8,  28. ])
>>> outer(A, [10, 20, 30])
array([[ 12.,  24.,  36.],
       [  2.,   4.,   6.],
       [ 20.,  40.,  60.]])
>>> A / 2.0
array([ 0.6,  0.1,  1. ])
```

These operations all create new arrays. You can also use `*=`, `+=`, *etc* for in-place operations, which will overwrite the existing array with new values.

The `:` symbol can be used to get rows, columns and ranges of an array. You can also pass lists of indices with `[]`. Note that these operations also return arrays.

```
>>> M[0,1]
2.0
>>> M[:,1]
array([ 2.,  5.,  8.])
>>> M[1:2,:]
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> M[[0,2],:]
array([[ 1.,  2.,  3.],
       [ 7.,  8.,  9.]])
>>> A[::-1]
array([ 2. ,  0.2,  1.2])
```

Also note that rank-1 arrays are sequences, not vectors. For vector operations, NumPy will automatically transpose rank-1 arrays as necessary (you can use `.T` as shorthand for `.transpose()`).

```
>>> all(M==M.T)
False
>>> all(A==A.T)
True
>>> dot(M, A)
array([  3.8,   8.9,  14. ])
>>> dot(M, A.T)
array([  3.8,   8.9,  14. ])
```

CREATING ARRAYS

You can create arrays with handy functions including `arange()` and `linspace()` (create arrays over ranges and intervals), as well as `zeros()` and `ones()` (create zero- or one-initialized arrays) and `eye()` (create the identity matrix). There are also more advanced tools, such as `fromfunction()` and `fromfile()`.

```
>>> arange(0, 8)
array([3, 4, 5, 6, 7])
>>> arange(3, 8, 2)
array([3, 5, 7])
>>> linspace(-20,20,5)
array([-20., -10.,   0.,  10.,  20.])
>>> ones((2,2))
array([[ 1.,  1.],
       [ 1.,  1.]])
>>> zeros(5, dtype=int)
array([ 0,  0,  0,  0,  0])
>>> eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> fromfunction(multiply, (3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  4.]])
```

You can change the shape (and rank) of an array (a -1 value indicates NumPy should use whatever value is required for the data):

```
>>> Z = arange(8)
>>> Z
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> Z.shape
(8,)
>>> Z.shape = (2,4)
>>> Z
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> Z.reshape(2,2,-1)
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
>>> Z[:,0,0]
array([0, 4])
```

UPCASTING

As you may have noticed in the examples, if you mix arrays of different data types, NumPy automatically performs *upcasting* for the return values. That is, the more precise type will be chosen:

```
>>> B = array([True, False, True], dtype=bool)
>>> F = array([0.1, 1., 10.], dtype=float)
>>> I = array([3, 4, 5], dtype=int)
>>> B * I
array([3, 0, 5])
>>> (B * I).dtype
dtype('int32')
>>> I * F
array([  0.3,   4. ,  50. ])
>>> (I * F).dtype
dtype('float32')
```

NumPy is intended to be relatively high-performance, so it only copies data when absolutely necessary. When you use assignments or slices, you create a new reference or array object (respectively), but it points at the same data. This can be confusing to new users, and is different from how Python handles list slices, so watch out!

COPYING ARRAYS

```
>>> X = arange(8)
>>> X
array([0, 1, 2, 3, 4, 5, 6, 7])
>>> Y = X
>>> Y[0] = -5
>>> X
array([-5,  1,  2,  3,  4,  5,  6,  7])    # !
>>> Z = X[2:5]
>>> Z[0] = 100
>>> X
array([ -5,   1, 100,   3,   4,   5,   6,   7])  # !!!
```

To make a copy of the data in an array, use `.copy()`.

```
>>> X = arange(8)
>>> Z = X[2:5].copy()
>>> Z[0] = 100
>>> Z
array([100,   3,   4])
>>> X
array([0, 1, 2, 3, 4, 5, 6, 7])
```

2.6 Building a link graph from HTML files

Now that we've learned a few of the basics, let's do some actual Python programming.

First, we're going to read several HTML files and extract all the links. Download the data from the course website: `cs.ubc.ca/~nando/` To make it easy, we're not going to worry too much about line breaks or proper HTML parsing right now.

```
>>> links = {}
>>> fnames = ['angelinajolie.html', 'bradpitt.html',
 'jenniferaniston.html', 'jonvoight.html',
 'martinscorcese.html', 'robertdeniro.html']

>>> for file in fnames:
...     links[file] = []
...     f = open(file)
...     for line in f.readlines():
...         while True:
...             p = line.partition('<a href="http://')[2]
...             if p=='':
...                 break
...             url, _, line = p.partition('\">')
...             links[file].append(url)
...     f.close()
```

`open()` creates a `file` object for reading. The string `partition()` method returns a list of three substrings: before, 'during', and after the partition string. If the partition string was not found, the second and third strings will be empty, and we can skip this line.

After this, we will have a dict called `links` for which the keys are filenames, and the values are the links in the files. Now we want to make this into a graph using the NetworkX library.

```
>>> import networkx as nx
>>> DG = nx.DiGraph()
>>> DG.add_nodes_from(fnames)
>>> edges = []
>>> for key, values in links.iteritems():
...     eweight = {}
...     for v in values:
...         if v in eweight:
...             eweight[v] += 1
...         else:
...             eweight[v] = 1
...     for succ, weight in eweight.iteritems():
...         edges.append([key, succ, {'weight':weight}])

>>> DG.add_edges_from(edges)
```

We have now told the `DiGraph` object about its edges and nodes. We can visualize the graph with the `matplotlib` plotting library:

MATPLOTLIB

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(9,9))
>>> pos=nx.spring_layout(DG,iterations=10)
>>> nx.draw(DG,pos,node_size=0,alpha=0.4,edge_color='r',
            font_size=16)
>>> plt.savefig("link_graph.png")
>>> plt.show()
```

If you have followed all the steps correctly, you should be able to see the plot of Figure 2.1 on your screen.

In the next part of this tutorial, we will be using this graph, so let's save it to disk. In Python, often the easiest way to write data is to use `pickle`, which can read and write most Python types.

PICKLE

```
>>> import cPickle as pickle
>>> pickle.dump(DG, open('DG.pkl','w'))
```

This will write the object out to disk in the file `DG.pkl`. You can reload it later, but you will also need to re-import the networkx module to interact with it.

```
>>> import cPickle as pickle
>>> import networkx as nx
>>> DG = pickle.load(open('DG.pkl'))
```

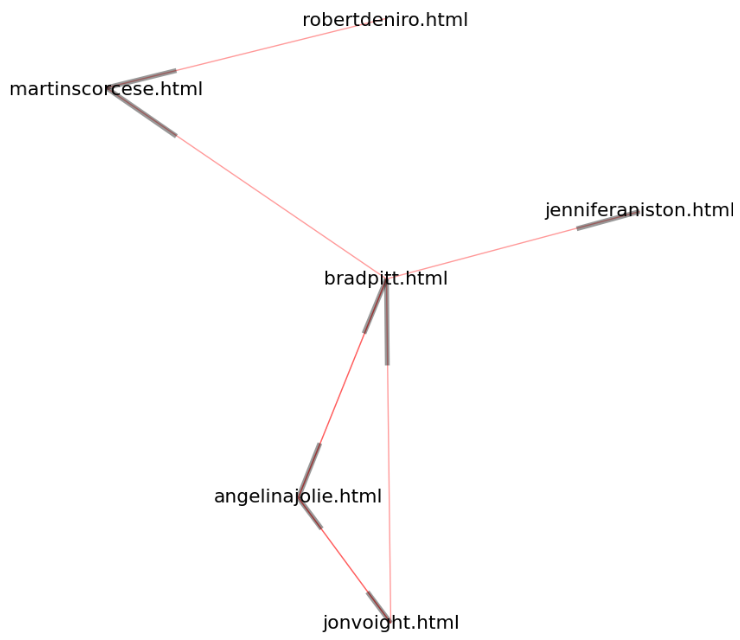

Figure 2.1 Webpage link-graph.

2.7 Ranking the webpages with pagerank

The webpage link-graph of Figure 2.1, where the nodes represent the webpages and the arrows the links between them, can be mapped to a *transition matrix* **T**. This mapping is actually fairly easy to implement using the Python data structures we have just learned, but you may need to look at the results of each step to understand how it works.

TRANSITION MATRIX

First, we make an empty transition matrix of size N_x-by-N_x, where N_x is the number of pages.

```
>>> from numpy import zeros
>>> NX = len(fnames)
>>> T = matrix(zeros((NX, NX)))
```

We need a way to match the filenames to the indices of the matrix. We will do this with a dict. We can populate a new dict by giving it (*key*, *value*) pairs. Since we want each key to map to an index of the key's position in the original list, we can use Python's `enumerate()` function, which iterates over a sequence, returning values and indices.

```
>>> f2i = dict((fn, i) for i, fn in enumerate(fnames))
>>> f2i
{'angelinajolie.html': 0,
 'bradpitt.html': 1,
 'jenniferaniston.html': 2,
 'jonvoight.html': 3,
 'martinscorcese.html': 4,
 'robertdeniro.html': 5}
>>> f2i['jonvoight.html']
3
```

Now, we can iterate over the DiGraph's adjacency data to fill in the transition matrix, using the predecessors as rows and the successors as columns.

```
>>> for predecessor, successors in DG.adj.iteritems():
...     for s, edata in successors.iteritems():
...         T[f2i[predecessor], f2i[s]] = edata['weight']
>>> T
matrix([[ 0.,  1.,  0.,  1.,  0.,  0.],
        [ 2.,  0.,  1.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.],
        [ 2.,  1.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  1.,  0.]])
```

Next, we add a matrix of uniform probability **E** to **T**:

$$\mathbf{L} = \mathbf{T} + \epsilon\mathbf{E}$$

where ϵ is a small scalar and **E** has entries $1/N_x$. **L** is then normalized so that its rows add up to 1; resulting in a new matrix **G**. That is

$$\sum_j \mathbf{G}_{i,j} = 1 \qquad \forall i.$$

NumPy contains a module named random, which has a function also called random, which returns random numbers between 0 and 1, for an array of given size and dimensions. We can use this to create the **L** and **G** matrices.

```
>>> from numpy.random import random
>>> from numpy import sum, ones
>>> epsilon = .01
>>> E = ones(T.shape)/NX
>>> L = T + epsilon * E
>>> G = matrix(zeros(L.shape))
>>> for i in xrange(NX):
...     G[i,:] = L[i,:] / sum(L[i,:])
```

MARKOV CHAIN
STOCHASTIC MATRIX

G is a *stochastic matrix*, also known as a *Markov chain* transition matrix. This class of matrices has been deeply studied in mathematics. For example, it is known that given any initial vector $\boldsymbol{\pi}$, whose entries add up to 1, successive multiplication by the matrix **G** results in a vector $\mathbf{p}$ whose entries remain invariant, and which add up to 1. In mathematical terms:

$$\lim_{N\to\infty} \boldsymbol{\pi}^T \mathbf{L}^N = \mathbf{p}^T$$

For this to happen, the *Perron-Frobenius Theorem* of linear algebra tells us that the stochastic transition matrix must obey the following properties:

PERRON-FROBENIUS THEOREM

1. *Irreducibility*: For any state (webpage), there is a positive probability of visiting all other states. That is, the matrix $\mathbf{L}$ cannot be reduced to block-diagonal form. This is equivalent to requiring that the transition graph be connected. If a webpage is disconnected from the graph, then the random surfer will never be able to find it by simply following links.
2. *Aperiodicity*: The Markov chain should not get trapped in cycles. Why? What happens when

$$\mathbf{G} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}?$$

   Try a few iterations with say $\boldsymbol{\pi} = (1/3,\ 2/3)$.

Our desire to satisfy these properties is what drove us to add a small random perturbation matrix $\epsilon\mathbf{E}$ to $\mathbf{T}$ in order to form the stochastic matrix $\mathbf{G}$. We want our algorithm to converge to the same $\mathbf{p}$ for any initial vector $\boldsymbol{\pi}$.

In the perhaps more intuitive web setting, the random perturbation can also be thought of as a way of "teleporting" to another webpage via a mechanism other than links (say, via personal bookmarks).

This is all wonderful mathematically, but you're probably asking yourself what is the meaning of $\mathbf{p}$? To answer this question, let us enter the world of information retrieval and search engines.

In information retrieval, we are interested in estimating the *relevance* of each webpage. To make the argument more concrete, it is reasonable to think of relevance as the probability of a webpage being visited by someone surfing the web randomly. In this model, the *random surfer* visits a webpage and clicks on one of the links in that page at random. Hence, if a page has more links to a second page, this second page is more likely to be selected by the random surfer.

INFORMATION RETRIEVAL RELEVANCE

RANDOM SURFER MODEL

Intuitively, if a person is clicking on links at random, it is more likely that she will end up at `cnn.com` than at `cs.ubc.ca`.

The popular search engine *Google* harnessed this intuition to design an automatic algorithm for ranking webpages called *pagerank*. Pagerank follows exactly the process we've described in this section. It computes the stationary vector $\mathbf{p}$ by successive vector-matrix multiplication (in sparse form of course because $\mathbf{G}$ has billions of rows

GOOGLE

PAGERANK

and columns!). The entries of **p** correspond to the probabilities of each webpage being visited by the random surfer.

In Python, pagerank proceeds as follows:

```
>>> PI = random(NX)
>>> PI /= sum(PI)
>>> R = PI
>>> for _ in xrange(100):
...     R = dot(R, G)
```

How quickly does this algorithm converge? What determines the rate of convergence? Again matrix algebra and spectral theory provide the answers, but we shall not explore these in this chapter.

Empirically, however, we can plot the individual entries of the rank vector on the vertical axis and the number of iterations on the horizontal axis. This allows us to see how the algorithm is converging and determine how many iterations are necessary.

```
>>> import matplotlib.pyplot as plt
>>> evolution = [dot(PI, G**i) for i in xrange(1, 20)]
>>> plt.figure()
>>> for i in xrange(NX):
>>>     plt.plot([step[0,i] for step in evolution],
                 label=fnames[i], lw=2)
>>> plt.draw()
```

Note that because `dot(PI, G**i)` returns a matrix, we must specify the row and column of `step`, even though the row is always 0.

Matplotlib is a powerful plotting library with many different tools. `plot()` creates line and scatter plots. The `label` parameter allows us to generate legends to interpret the plot. We can also add titles and axis labels. The syntax is (intentionally) very similar to MATLAB's plotting syntax.

```
>>> plt.title('rank vs iterations')
>>> plt.xlabel('iterations')
>>> plt.ylabel('rank')
>>> plt.legend()
>>> plt.draw()
```

The resulting plot is shown in Figure 2.2. Do the ranks of our celebrities seem reasonable given their link-graph?

What are the eigenvalues of $\mathbf{G}$ and $\mathbf{G}^{1000}$? What is happening to the eigenvalues of $\mathbf{G}^N$ as N increases? How do you think this might relate to the convergence result we mentioned earlier in this section?

```
>>> from numpy import linalg
>>> linalg.eigvals(G**1000)
array([  0.00000000e+00, 1.00000000e+00, 3.20923843e-17,
        -3.59103089e-33, 6.44186704e-34, 0.00000000e+00])
```

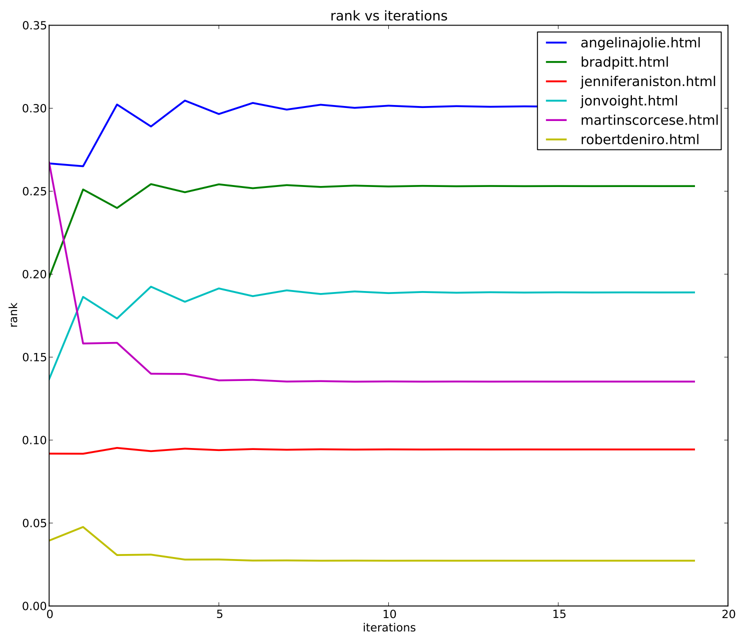

Figure 2.2 Pagerank applied to our celebrities link-graph.

It turns our that the *Perron Frobenius Theorem*, whose proof is outside the scope of this book, provides an answer to these questions. In particular, it states that **G** has one eigenvalue that is equal to 1. All other eigenvalues are strictly less than 1 in absolute value. What is the eigenvector corresponding to the eigenvalue 1? As we increase the powers of **G**, which eigenvalue (other than 1) will take longer to vanish?

Now let us turn our attention back to information retrieval. For the keyword query "acting", what pages (in order and using exact word matching) does our simple search engine return?

KEYWORD QUERY

To find this out, we will create a *reverse index* – this is simply a mapping that, for a given keyword, returns a list of files in which the keyword appears, and counts for that keyword. This is another situation where Python's data types help us out. To create the reverse index, we will create a dict, in which each key is a keyword token, and each value is *another* dict, which maps files to counts of the keyword. (We could make this even more concise by using the `defaultdict` type from the `collections` module, but it makes the code slightly harder to follow.)

REVERSE INDEX

```
>>> revind = {}
>>> fnames = ['angelinajolie.html', 'bradpitt.html',
  'jenniferaniston.html', 'jonvoight.html',
  'martinscorcese.html', 'robertdeniro.html']
>>> for fname in fnames:
...     for line in open(fname).readlines():
...         for token in line.split():
...             if token in revind:
...                 if fname in revind[token]:
...                     revind[token][fname] += 1
...                 else:
...                     revind[token][fname] = 1
...             else:
...                 revind[token] = {fname: 1}
```

The `split()` string method splits a string on the spaces, and returns a sequence of substrings.

We can now query the reverse index and then sort the results.

```
>>> revind['film']
{'angelinajolie.html': 3,
 'bradpitt.html': 3,
 'jenniferaniston.html': 1,
 'jonvoight.html': 1,
 'martinscorcese.html': 5,
 'robertdeniro.html': 1}
```

Now, we wish to sort the results. Python's `sorted()` function has an argument `key=`, which handles cases where sorting is ambiguous, such as dict items (that is, we could be sorting on the keys, or the values and need to tell it which). Without getting into too many details right now, `key` asks for a function, which takes as its argument an element of the sort sequence, and which returns a value for the object which corresponds to the one we actually want to sort on. Conveniently, that's exactly what `getPageRank()` does.

```
>>> def getPageRank(fname):
...     return R[0,f2i[fname]]
```

This will return the pagerank of the filename, which is what we actually want to sort on. Note that because the function is defined in the command-line environment, functions have access to all the variables we have already declared in that environment, such as `R` and `f2i`. Functions and methods defined in modules do not (usually) have access to those variables.

Now, we can pass `getPageRank` as the value for `sort()`'s `key` parameter. We also pass it the `reverse=True` parameter, which tells `sort()` to sort from highest value to lowest (by default, it sorts low-to-high)...

```
>>> result = revind['film'].keys()
>>> result
['robertdeniro.html',
 'angelinajolie.html',
 'martinscorcese.html',
 'jonvoight.html',
 'bradpitt.html',
 'jenniferaniston.html']
>>> sorted(result, key=getPageRank, reverse=True)
['angelinajolie.html',
 'bradpitt.html',
 'jonvoight.html',
 'martinscorcese.html',
 'jenniferaniston.html',
 'robertdeniro.html']
```

...and we get back our results in a very Google-esque order, with the highest-ranked page at the top, and the lowest-ranked at the bottom.

If you've been typing these commands, you've just finished implementing your first search engine. Congratulations!

This is now a good time to try some exercises to practice Python and learn more about all the exciting things you can do with this language. The exercises will also allow you to revise fundamentals of probability and linear algebra.

In hte coding exercises, fill in Python code wherever you see the `???` symbol. Submit your code and plots.

Depending on your interface and settings, you may find that figures are not updated after plotting. If this is the case, use the `draw()` function. If you find you need to clear your figure while coding, you can use the `clf()` command.

```
>>> from matplotlib.pyplot import draw, clf
>>> figure()
>>> plot (...)
>>> draw()
>>> clf()
>>> plot (...)
>>> draw()
```

Exercises

2.1 **Eigenvalues and eigenvectors.**

Compute the eigenvalues and eigenvectors of the following matrix by hand and using NumPy:

$$\mathbf{A} = \begin{pmatrix} -2 & 2 & -3 \\ 2 & 1 & -6 \\ -1 & -2 & 0 \end{pmatrix}$$

Hand in the NumPy code as well as your derivation.

2.2 **Plotting one-dimensional Gaussians.**
For a one-dimensional Gaussian distribution with a mean μ and variance σ, the probability density at a point, x, is purely a measure of the point's distance from the mean:

GAUSSIAN PROBABILITY DENSITY FUNCTION (PDF)

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

This function is actually already present in the `scipy.stats` module, but we are going to reimplement it as our own function:

```
>>> from math import sqrt, exp, pi
>>> def gaussianPDF(mu, sigma, x):
...     return 1. / sqrt(2. * pi) * ???
```

We can now plot a one-dimensional Gaussian distribution with $\mu = 0, \sigma = 1$, also called a *standard normal distribution*.

```
>>> from matplotlib.pylab import *
>>> from numpy import arange
>>> samples = arange(-5., 5., .05)
>>> figure(1)
>>> plot(samples, [??? for x in samples], lw=2.)
>>> grid()
```

Return a plot showing the standard normal distribution. Also show Gaussians with two other values of μ and σ, over the same range.

2.3 **Plotting two-dimensional Gaussians.**
Now, we're going to see how we can use NumPy arrays to plot two-dimensional Gaussians. We define $\boldsymbol{\mu}$ as a 2-element array of the means of the Gaussian in the two dimensions. Pretty straightforward. σ, however, becomes a 2-by-2 symmetric covariance matrix, and we acknowledge its new status with a new symbol: $\boldsymbol{\Sigma}$. The diagonal entries of $\boldsymbol{\Sigma}$ show how much the Gaussian varies along the axis, and the other entries show how the two dimensions interact. You can think of the diagonal entries as the "width" of the Gaussian, and the off-diagonals as the "orientation".

MULTIVARIATE GAUSSIAN

The PDF of a 2-dimensional Gaussian is:

$$p(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{|2\pi\boldsymbol{\Sigma}|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)$$

Let's implement this function in Python, too. We're going to import the NumPy version of `exp()`, which is designed to work with arrays and matrices. To multiply arrays and matrices in NumPy, use `dot()`.

```
>>> from numpy import exp, det, dot
>>> def gaussianPDF_2D(mu, Sigma, x):
...     d = 1. / sqrt(det(2. * pi * Sigma)) * ???
...     return float(d)
>>> Sigma = matrix('1. 0.; 0. 1.')
>>> mu = array([0., 0.])
>>> gaussianPDF_2D(mu, Sigma, array([0., 0.]))
???
```

Two common ways of plotting 2-dimensional functions are *contour* and *surface* plots. In both cases, we use special `meshgrid` arrays, which are simply regular 2-D arrays created from 1-D arrays, designed to make it easy to evaluate and plot a grid.

```
>>> from numpy import meshgrid, zeros
>>> x = arange(-5., 5., .05)
>>> y = arange(-5., 5., .05)
>>> X, Y = meshgrid(x, y)
>>> x.shape
(200,)
>>> X.shape
(200, 200)
>>> Z = zeros(X.shape)
>>> nx, ny = X.shape
>>> for i in xrange(nx):
...     for j in xrange(ny):
...         Z[i,j] = gaussianPDF_2D(mu, Sigma,
...                     array([X[i,j], Y[i,j]]))

>>> figure(2)
>>> contour(X, Y, Z)
```

3-D plotting of 2-D functions isn't fully integrated into Matplotlib (and you must be running version 0.99 or higher), so the syntax isn't quite as elegant as `contour()`.

```
>>> from mpl_toolkits.mplot3d import Axes3D
>>> fig = figure(3)
>>> ax = Axes3D(fig)
>>> ax.plot_surface(X, Y, Z)
```

Now, given two different Gaussians, plot the *sum* of the two.

```
>>> mu1 = array([1.5, 2.])
>>> Sigma1 = matrix('2. .1; .1 2.')
>>> mu2 = array([0., -1.])
>>> Sigma2 = matrix('1. -.5; -.5 1.')
>>> Z2 = zeros(X.shape)
>>> nx, ny = X.shape
>>> for i in xrange(nx):
...     for j in xrange(ny):
...         Z2[i,j] = ???
>>> ???
```

2.4 **Image processing.**
One of the great things about NumPy/SciPy is that it can work with all kinds of array data, such as image files, which, after all are just arrays of numbers. There are several powerful image-processing Python toolkits available, but we can already get started with NumPy.
First, get a (colour) .PNG image file from your computer or the internet. Now, you can load the image data into Python (where `PATHTOIMAGE` is the path to the image file on your computer).

```
>>> from numpy import imread, imshow
>>> img = imread('PATHTOIMAGE')
>>> figure()
>>> imshow(img)
```

Now we can create a gallery of altered images using `subplot()`, a function for putting several plots into a single one.

```
>>> from matplotlib.pyplot import subplot
>>> from scipy import ndimage
>>> figure()
>>> subplot(2,2,1)
>>> imshow(img)
>>> subplot(2,2,2)
>>> imshow(ndimage.rotate(img, 90))
>>> subplot(2,2,3)
>>> nored = img.copy()
>>> nored[:,:,0] = zeros(nored[:,:,0].shape)
>>> imshow(nored)
>>> subplot(2,2,4)
>>> onlyred = ???
>>> imshow(onlyred)
```

The third subplot shows the image with the red channel set to all-zero. For the last subplot, show an image where all channels *except* red are all zero.

NLTK

2.5 Text processing with NLTK

NATURAL LANGUAGE PROCESSING

The Natural Language Toolkit is a Python toolkit for research in natural language (*ie*, everyday speech and writing, as opposed to computer languages). It has many powerful features and a large set of documents for machine learning, but right now, we're just going to use it to improve our reverse index.

TOKENIZATION

To start, we are going to use a smarter tokenizer. In the Pagerank exercise, we use the Python `split()` method as our tokenizer. This divides a line into words based on spaces only. While it will successfully find most words, we also get tokens with punctuation stuck to them.

```
>>> x = 'This (sentence) has punctuation.'
>>> x.split()
['This', '(sentence)', 'has', 'punctuation.']
```

NLTK has a number of tokenizers – we'll be using a proper word tokenizer, which is the NLTK recommended default.

```
>>> from nltk import word_tokenize
>>> word_tokenize(x)
['This', '(', 'sentence', ')', 'has', 'punctuation', '.']
```

STEMMING

Another important tool for natural language processing is *stemming* – this involves turning words with various prefixes and suffixes – "running", "runs", *etc* – into a common form. To accomplish this, we will use a common stemming algorithm, the Porter stemmer.

```
>>> from nltk.stem.porter import PorterStemmer
>>> stemmer = PorterStemmer()
>>> for word in ['run', 'running', 'runs', 'ran']:
...     print word, '->', stemmer.stem(word)
run -> run
running -> run
runs -> run
ran -> ran
```

Note that "ran" is not converted to "run". Stemming is purely lexical – it has no method for knowing common roots, just letters. (This requires *lemmatization*, which NLTK also supports, but which we won't be getting into right now.) LEMMATIZATION

In information retrieval, an important measure of the relevance of a word to identifying a document is *tf-idf* (term frequency-inverse document frequency). This is a score for each document-word pair in a corpus (a collection of documents) that is high when the word appears frequently in the document but infrequently overall, and low when it appears it many documents – if we want to identify the topic of a document, a word like "Vancouver" is more relevant than "the". CORPUS

The *term frequency* of word w in document d is equal to the total number of times w appears in d, divided by the total number of words in d.

The *inverse document frequency* of word w is based on the logarithm of the inverse frequency of the word in the corpus. If the number of documents in the corpus is D and the number of documents the word appears in is D_w, then

$$idf_w = \log \frac{D}{1 + D_w}$$

and for w and d, we can compute

$$tfidf_{w,d} = tf_{w,d} idf_w$$

Now, we will use the things we have just learned to compute tf-idf on the web pages we used in the Pagerank exercise. First, create a reverse index like the one we used above, except using the NLTK tokenizer and stemmer.

Then, write a `tfidf()` function that prints, for a given word, the tf-idf score for the *stem* of the word in each of the documents it appears in:

```
>>> from __future__ import division
>>> from math import log
>>> fname_total_words = {}
>>> revind = {}
>>> fnames = ['angelinajolie.html',
        'bradpitt.html',
        'jenniferaniston.html',
        'jonvoight.html',
        'martinscorcese.html',
        'robertdeniro.html']
>>> for fname in fnames:
...     fname_total_words[fname] = 0
...     for line in open(fname).readlines():
...         ??? revind ???
...         ??? fname_total_words ???

>>> def tfidf(revind, fname_total_words, word):
...     wstem = stemmer.stem(word)
...     if wstem != word:
...         print "word '%s' stemmed to '%s'" % (word,
    wstem)
...     idf = log(len(fnames) / (1 + len(revind[wstem])))
...     for fn, count in revind[wstem].iteritems():
...         tfidf = idf * ???
...         print "\tTF-IDF for '%s' = %f" % (fn, tfidf)

>>> tfidf(revind, fname_total_words, 'acting')
 ???
>>> tfidf(revind, fname_total_words, 'awards')
 ???
>>> tfidf(revind, fname_total_words, 'and')
 ???
```

`fname_total_words` is a dict with filenames as keys and total word counts as values.

The `from __future__ import division` command makes the division of integers return a float. As the name implies, this will be the default in a future version of Python.

2.6 **Matrix eigen-decomosition.**
Suppose the matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ has n linearly independent eigenvectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Define the matrix $\mathbf{Q}$ having these vectors as columns, i.e. $\mathbf{Q} = [\mathbf{x}_1 \quad \mathbf{x}_2 \quad \ldots \quad \mathbf{x}_n]$. Let $\mathbf{D}$ be a diagonal matrix with the eigenvalues λ_i in the diagonal. Show that

$$\mathbf{A} = \mathbf{QDQ}^{-1}$$

2.7 **Diagonalization of real symmetric matrices.**

1. Prove that a real symmetric matrix, $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n \times n}$, has real eigenvalues.
2. Prove the fact that real symmetric matrices with distinct eigenvalues have orthogonal eigenvectors.
3. Use the two previous facts to show that in this case $\mathbf{A}$ admits the following decomposition

$$\mathbf{A} = \mathbf{QDQ}^T$$