The Google PageRank Algorithm in 126 Lines of Python

Many people were surprised by the simplicity of the math underlying the google PageRank algorithm, and the ease with which it seemed to be efficiently implementable.

Many people were surprised by the simplicity of the math underlying the google PageRank algorithm, and the ease with which it seemed to be efficiently implementable.

Reading How Google Finds Your Needle in the Web's Haystack I was surprised by the simplicity of the math underlying the google PageRank algorithm, and the ease with which it seemed to be efficiently implementable. Being able to do a google-style ranking seems useful for a wide range of cases, and since I had wanted to take a look at python for numerics for some time, I decided to give it a shot (note that there already exists a python implementation using the numarray module).
Note that PageRank™ is a trademark of Google and that the described algorithm is covered by numerous patents. I hope the powers that be will nevertheless ignore/forgive my insolence.

The Math

A recent article on the AMS featurecolumn neatly described the google PageRank algorithm, showing in a few relatively easy steps that for a set of N linked pages it boils down to finding the N-dimensional (column) vector I of ``page relevance coefficients'' which one can formulate to be (see the above article) the principal eigenvector (*i.e.* the one corresponding to the largest eigenvalue) of the google matrix **G** defined as
$\mathbf{G} = \alpha(\mathbf{H} + \mathbf{A}) + (1 - \alpha)\ \mathbf{1}_N$.   (1)
Here, α is some parameter between 0 and 1 (typically taken to be 0.85). For the entry specified by row i and column j we define $\mathbf{H}_{ij} = 1 / l_j$ if page j links to page i (and l_j is the total number of links on page j), and 0 otherwise, such that the ``relevance'' of page i is
$I_i = \Sigma_j\ \mathbf{H}_{ij}\ I_j$,   (2)
corresponding to the number of links pointing to each page, weighted by the relevance of the source page divided by the number of links emanating from the source page. Similarly, we define $\mathbf{A}_{ij} = 1 / N$ if j is a page with no outgoing links, and 0 otherwise. So each page has a total of 1 outgoing ``link weights'', *i.e.* the sum over the elements of each column of $\mathbf{H} + \mathbf{A}$ is one (it is a stochastic matrix). Finally, $\mathbf{1}_N$ is defined to be an N x N matrix with all elements

equal to 1 / N (*it is not the identity matrix*), and is therefore also stochastic. Similarly, **G** is stochastic. This latter statement is important, because for stochastic matrices the largest eigenvalue is 1, and the corresponding eigenvector can accordingly be found using the power method, which will turn out to be very efficient in this case.

> Summarizing, **G** (a finite markov chain) may be interpreted to model the behaviour of a user who stays at each page for the same amount of time, then either (with probability α) randomly clicks a link on this page (or goes to a random page if no outgoing links exist), or picks a random page off of the www (with probability 1 - α).

Finally, the power method relies on the fact that for eigenvalue problems where the largest eigenvalue is non-degenerate and equal to 1 (which is the case), one can find a good approximate for the principal eigenvector via an iterative procedure starting from some (arbitrary) guess I^0:

$I = I^k = \mathbf{G}^k I^0$, $k \to \infty$.   (3)

The Algorithm

First off, it may be noted that the current number of publically accessible pages on the www (*i.e.* N) is estimated to be somewhere in the billions (2006) and rapidly growing. So for (3) to be usable, it must be highly efficient. Obviously, the repeated matrix multiplication in (3) is ripe with potential optimizations, but we'll just focus on the optimizations pointed out in the article inspiring this essay, namely that

- **H** is extremely sparse (on the order of 10 links per page), *i.e.* the product $\mathbf{H}I$ can be obtained with linear effort in both the number of operations and the storage requirements. Note however, that in order to perform this operation efficiently, we first need to transpose **H**, *i.e.* formulate the link matrix in terms of incoming links, rather than outgoing links.
- All column vectors are identical in either **A** or $\mathbf{1}_N$, so it is sufficient to compute just one row-vector product (*i.e.* $\mathbf{A}_i I$ and $\mathbf{1}_{N,i} I$).
- **A** is also sparse (there are few pages with no links), and the non-zero columns all have a value of 1 / N (for those pages with no outgoing links), so the elements of $\mathbf{A}_i I$ are not only all equal but are the sum over the entries of I corresponding to those pages with no links, divided by N.
- Finally, the elements of $\mathbf{1}_{N,i} I$ are all equal and are equal to the sum over all elements of I divided by N.

Similarly, convergence can be checked with an effort of (at most) $\boldsymbol{O}(N)$ and the algorithm is known to converge rather quickly with a choice of α close to 1. The average deviation <δ> of the elements of *I* should be a reasonable way to check convergence every *m* iterations:

$<\delta>^2 = (I^k)^t I^{k+m}/N^2$. (4)

The Code

All of the following is done in python, with the numpy library, so don't forget to

```
from numpy import *
```

Full source code.

First, we need to transform the matrix of outgoing links into one of incoming links, while preserving the number of outgoing links per page and identifying leaf nodes:

```
def transposeLinkMatrix(
outGoingLinks = [[]]
):
"""
Transpose the link matrix. The link matrix contains the pages each page points to.
However, what we want is to know which pages point to a given page. However, we
will still want to know how many links each page contains (so store that in a separate array),
as well as which pages contain no links at all (leaf nodes).

@param outGoingLinks outGoingLinks[ii] contains the indices of pages pointed to by page ii
@return a tuple of (incomingLinks, numOutGoingLinks, leafNodes)
"""

nPages = len(outGoingLinks)
# incomingLinks[ii] will contain the indices jj of the pages linking to page ii
incomingLinks = [[] for ii in range(nPages)]
# the number of links in each page
numLinks = zeros(nPages, int32)
# the indices of the leaf nodes
leafNodes = []

for ii in range(nPages):
if len(outGoingLinks[ii]) == 0:
leafNodes.append(ii)
else:
numLinks[ii] = len(outGoingLinks[ii])
# transpose the link matrix
for jj in outGoingLinks[ii]:
```

```
incomingLinks[jj].append(ii)

incomingLinks = [array(ii) for ii in incomingLinks]
numLinks = array(numLinks)
leafNodes = array(leafNodes)

return incomingLinks, numLinks, leafNodes
```

Next, we need to perform the iterative refinement. Wrapping it up in a generator to hide the state of the iteration behind a function-like interface. Single-precision should be good enough.

```
def pageRankGenerator(
At = [array((), int32)],
numLinks = array((), int32),
ln = array((), int32),
alpha = 0.85,
convergence = 0.01,
checkSteps = 10
):
"""
Compute an approximate page rank vector of N pages to within some convergence factor.
@param At a sparse square matrix with N rows. At[ii] contains the indices of pages jj linking to ii.
@param numLinks iNumLinks[ii] is the number of links going out from ii.
@param ln contains the indices of pages without links
@param alpha a value between 0 and 1. Determines the relative importance of "stochastic" links.
@param convergence a relative convergence criterion. smaller means better, but more expensive.
@param checkSteps check for convergence after so many steps
"""

# the number of "pages"
N = len(At)

# the number of "pages without links"
M = ln.shape[0]

# initialize: single-precision should be good enough
iNew = ones((N,), float32) / N
iOld = ones((N,), float32) / N
```

```
done = False
while not done:

# normalize every now and then for numerical stability
iNew /= sum(iNew)

for step in range(checkSteps):

# swap arrays
iOld, iNew = iNew, iOld

# an element in the 1 x I vector.
# all elements are identical.
oneIv = (1 - alpha) * sum(iOld) / N

# an element of the A x I vector.
# all elements are identical.
oneAv = 0.0
if M > 0:
oneAv = alpha * sum(iOld.take(In, axis = 0)) * M / N

# the elements of the H x I multiplication
ii = 0
while ii < N:
page = At[ii]
h = 0
if page.shape[0]:
h = alpha * dot(
iOld.take(page, axis = 0),
1. / numLinks.take(page, axis = 0)
)
iNew[ii] = h + oneAv + oneIv
ii += 1

diff = iNew - iOld
done = (sqrt(dot(diff, diff)) / N < convergence)

yield iNew
```

Finally, we might want to wrap up the above behind a convenience interface like so:

```
def pageRank(
linkMatrix = [[]],
alpha = 0.85,
convergence = 0.01,
checkSteps = 10
):
"""
Convenience wrap for the link matrix transpose and the generator.
"""
incomingLinks, numLinks, leafNodes = transposeLinkMatrix(linkMatrix)

for gr in pageRankGenerator(incomingLinks, numLinks, leafNodes,
alpha = alpha,
convergence = convergence,
checkSteps = checkSteps):
final = gr

return final
```

The Validation

A Web Consisting of Two Pages with One Link

Given a link graph consisting of two pages, with page one linking to page two, one would expect an eigenvector of {1/3, 2/3}, if α is 1, and an eigenvector of {0.5, 0.5}, if α is zero. This is because page 2 is a leaf node (*i.e* it implicitly links to all pages), while page 1 links to page 2. Page 1 therefore has one link pointing to it, page 2 has two. The overall system contains three ``links''. We can test this using this simple script:

```
#!/usr/bin/env python

from pageRank import pageRank

# a graph with one link from page one to page two
links = [
                [1],
                []
        ]

print pageRank(links, alpha = 1.0)
```

which affords

```
./test.py
[ 0.33349609  0.66650391]
```

Fair enough. Changing α to 0.0, we find:

```
./test.py
[ 0.5  0.5]
```

Which is fine.

A Web Consisting of Just One Circle

Similarly, for a web consisting of just one circle (*i.e.* page *n* links to page *n+1*), we should find equal page rankings for all pages (independent of α). We do this by performing the following change in the above script:

```
#!/usr/bin/env python

from pageRank import *

# a graph with one link from N to page N + 1
links = [
                [1],
                [2],
                [3],
                [4],
                [0]
        ]

print pageRank(links, alpha = 1.0)
```

and find for α equal to 1.0:

```
./test.py
[ 0.2  0.2  0.2  0.2  0.2]
```

and for α equal to 0.0:

```
./test.py
[ 0.2  0.2  0.2  0.2  0.2]
```

A Web Consisting of Two Circles

```
#!/usr/bin/env python

from pageRank import *
```

```
# a graph with one link from N to page N + 1
links = [
                [1, 2],
                [2],
                [3],
                [4],
                [0]
        ]

print pageRank(links)
```

we find

```
[ 0.2116109   0.12411822  0.2296187   0.22099231  0.21365988]
```

We find that the relevance of the second page has decreased, because the first page also links to the third page. The third page has highest relevance, and the relevance decreases as one goes on to the fourth, fifth, and back to the third page. This makes sense!

A Quick Note on Performance

The above code was not optimized beyond the basic tweaks necessary to make it scale. A quick performance check might still be in order, so let's just make a large ring:

```
#!/usr/bin/env python

from pageRank import *
from random import randint

# a graph with one link from page one to page two
links = [[ii + 1] for ii in range(100000)]
links.append([0])
print "starting"
print pageRank(links)
print "stopping"
```

and running that on my 1.83 GHz MacBook, gives

```
time ./test_large_ring.py
starting
[  1.00007992e-05   1.00007992e-05   1.00007992e-05 ...,   1.00007992e-05
   1.00007992e-05   1.00007992e-05]
stopping

real    0m30.586s
user    0m30.139s
sys     0m0.338s
```

where (based on the printouts) the most time is spent in the iteration circle. The overall performance seems quite nice. Though it may obviously vary as one goes to more complicated link graphs. But it should certainly be appropriate to perform rankings of datasets with tens of thousands of links on very basic hardware.