CS 111: Homework 6: Due by 11:59 pm Wednesday, February 26, 2020

Submit your paper as one PDF file. There's only one problem on this week's assignment.

1. In class, we computed the PageRank ordering from the adjacency matrix of a directed graph by applying scipy's built-in `spla.eig()` function. The method we used doesn't work for very large graphs/matrices, because it forms a completely dense n-by-n matrix M, which requires $O(n^2)$ memory to store and $O(n^3)$ time to run `spla.eig()`. In this assignment you will write a python code that works for much larger graphs, using the power method to find the eigenvector, without ever forming a dense matrix. You will start with a program that I wrote, which uses the power method but still forms the dense matrix M, and you will modify it so that it doesn't need the dense matrix.

1 Code and data

The lecture files on the GitHub site for Homework 6,

`https://github.com/ucsb-cs111/w20-lecture-files/tree/master/homework_files/h06,`

include several files that you will use for this assignment:

- `cs111/pagerank.py` contains the function `cs111.pagerank1()`), which is the python code you'll start from.
- `PageRankEG1.npy` and `PageRankEG2.npy`, which are the small graphs (as dense adjacency matrices) I used as examples in class.
- `PageRankEG3.npy`, which is the graph of the 500-node Harvard web crawl.
- `PageRankEG3.nodelabels`, which lists the 500 Harvard site names.
- `PageRankEG*.npz`, which are the same graphs as the three above, but are stored as scipy `csr_sparse` matrices. You can use these to test your sparse code.
- `webGoogle.npz`, which is the graph of a web crawl of about 900,000 pages.
- `webGoogle.notes`, which is a text file with some info about the big graph.

Here is the result of running `cs111.pagerank1()` on matrix `PageRankEG1`:

```
In:
    E = np.load('PageRankEG1.npy')
    r, v = cs111.pagerank1(E, return_vector = True)
    print('r =', r)
    print('v =', v)

Out:
    Dominant eigenvalue is 1.000000 after 19 iterations.

    r = [0 2 3 1]
    v = [0.69648305 0.26828106 0.54477799 0.38230039]
```

And here are the ten top-ranked pages in the Harvard web crawl:

```
In:
    E = np.load(’PageRankEG3.npy’)
    sitename = open(’PageRankEG3.nodelabels’).read().splitlines()
    r, v = cs111.pagerank1(E, return_vector = True)
    print(’r[:10] =’, r[:10])
    print()
    for i in range(10):
        print(’rank %d is page %3d: %s’ % (i, r[i], sitename[r[i]]))

Out:
    Dominant eigenvalue is 1.000000 after 56 iterations.

    r[:10] = [  0   9  41 129  17  14   8  16  45  12]

    rank 0 is page   0: http://www.harvard.edu
    rank 1 is page   9: http://www.hbs.edu
    rank 2 is page  41: http://search.harvard.edu:8765/custom/query.html
    rank 3 is page 129: http://www.med.harvard.edu
    rank 4 is page  17: http://www.gse.harvard.edu
    rank 5 is page  14: http://www.hms.harvard.edu
    rank 6 is page   8: http://www.ksg.harvard.edu
    rank 7 is page  16: http://www.hsph.harvard.edu
    rank 8 is page  45: http://www.gocrimson.com
    rank 9 is page  12: http://www.hsdm.med.harvard.edu
```

The routine you write, `pagerank2()`, should be able to duplicate these results, and should also run correctly on the big web graph. If you run `cs111.pagerank1()` on the big web graph, Jupyter will either just hang or complain that it's out of memory. On my 4-year-old MacBook, my own version of `pagerank2()` takes about 6 seconds to run on the big web graph. (That's .35 seconds to load it from disk and 5.41 seconds to analyze it. Interestingly, `%time` says it used 9.98 seconds of run time, which means that under the covers numpy is doing a pretty good job of using the two cores in my laptop in parallel.)

2 The power method with a dense matrix

Let's look carefully at `cs111.pagerank1()`. The first few lines of code convert the matrix to a dense array if necessary and check the input for validity. Notice the sneaky test in the second `assert`, which verifies that every entry of `E` is either 0 or 1 without using an explicit loop or creating any new big matrices.

The section following comment (1.) fills in the empty columns of the matrix, which correspond to nodes with no outgoing links, by linking them to every other node in the graph. Note the line `F[j,j] = 0` that prevents a node from being linked to itself. The matrix `F` may have a lot more nonzeros than `E`; in your `pagerank2()`, you don't want to compute `F` explicitly.

The next few lines scale the matrix to make it column stochastic, and then create the matrix `M` that represents choosing a new node uniformly at random 15% of the time. Matrix `M` is completely dense, so you certainly don't want to compute it explicitly in `pagerank2()`.

Finally, the loop uses the power method to find the largest eigenvalue of `M` (which we know is equal to 1 by the Perron-Frobenius theorem) and its associated eigenvector (which gives the PageRank ratings).

The loop just multiplies the vector `v` by `M` repeatedly, rescaling it after each multiplication to have norm 1. The loop stops when the vector changes by less than a tolerance that defaults to 10^{-6}, or when the maximum number of iterations is reached.

The next couple of lines verify that (as promised by the Perron-Frobenius theorem) all the elements of the dominant eigenvector have the same sign, and make that sign positive. (Note that $-v$ is an eigenvector whenever v is.)

Finally, we compute the ranking permutation by sorting the eigenvector.

3 Getting rid of the dense matrix

Your `pagerank2()` should not compute any of the matrices `F`, `A`, `S`, or `M`. The question is, then, how do you get the effect of the line "`v = M @ v`"? You can use the fact that, mathematically, $M = (1 - m)A + mS$, so you can get the effect of multiplying a vector by M if you can multiply it both by A and by S. Given a vector v, what vector is Sv? How can you compute that vector without forming S?

Similarly, you can use the fact that $A = (E + F)/\text{sum}(E + F))$ to figure out how to compute Av from v by multiplying a suitable vector w only by the matrices E and F. In the end, the only matrix you actually need to multiply by is E.

4 What experiments to do

Write and debug a python function `pagerank2()` that has exactly the same input and outputs as `cs111.pagerank1()`, but forms no large matrices besides its input matrix `E`. Verify that your code gets the right results on the small examples and the Harvard crawl. You can load those examples as sparse matrices by using `E = scipy.sparse.load_npz('matrixname.npz')` instead of `E = np.load('matrixname.npy')`.

Run your code on the big web graph, timing it in Jupyter with `%time`. You should separately time loading the matrix from the `.npz` file on disk and computing the rankings with `pagerank2()`. What is the largest element in the PageRank vector? What is the smallest? Make a histogram of the logarithms of the elements of the eigenvector, which are the "importance" ratings.

Which is faster on the Harvard crawl, `pagerank2()` using `PageRankEG3.npz` or `cs111.pagerank1()` using `PageRankEG3.npy`?

How does the running time of your `pagerank2()` compare with the running time of `spla.eigs(E)` on the original web graph? (Of course they doesn't compute the same thing, but it's interesting to see how fast scipy's eigensolver is on a sparse matrix that size.) On my MacBook, `pagerank2()` is faster.

5 What to turn in

Include all of the following in your report:

- Your python source code `pagerank2()`, and any other python code you wrote.
- Jupyter output from your code duplicating the results in Section 1 above.
- Jupyter output from your code running on the big web graph, with timings from `%time`, and the largest and smallest elements of v. (Don't print out the values of r and v for this one!)
- Your histogram, formatted as nicely as you can.