Lab 5: PageRank

`CAS CS 132`: *Geometric Algorithms*

Due April 16, 2026 by 8:00PM

In this lab, you will be implementing *PageRank*, the algorithm behind the early days of Google Search.

1 Introduction

The idea behind pagerank is simple: the internet is a network that a "random surfer" can navigate. Websites can be ranked by how likely our surfer will end up at them in the long term. This is given by the *steady-state distribution* of the Markov chain determined by the internet.

In more detail: the internet can be represented by a stochastic matrix $A \in \mathbb{R}^{n \times n}$ where n is the total number of websites (roughly one billion as of now). The ith column $\mathbf{a}_i = [a_1 \ \dots \ a_n]^T$ of A is a probability vector where a_j is nonzero exactly when website i has a *link* to website j; furthermore, $a_j = \frac{1}{k}$ where k is the number of links on website i. In other words, our surfer is equally likely to follow any link on website i.

Suppose that our random surfer is equally likely to start at any website on the internet. Then the starting distribution of our surfer is $\mathbf{u} = [u_1 \ \dots \ u_n]^T \in \mathbb{R}^n$, where $u_i = \frac{1}{n}$ for all i from 1 to n. Our surfer's distribution in the long term is $\lim_{k \to \infty} A^k \mathbf{u}$, i.e., the steady state distribution of A. Therefore, in order to determine the most important website on the internet[1] we just need to determine the steady-state distribution $\mathbf{s}$ of A (i.e., the eigenvector with eigenvalue 1, and then **determine website whose corresponding entry in s is largest.**

We've seen how to determine this eigenvector, and numpy has a convenient function for finding eigenvectors called `numpy.linalg.eig`. However, a one billion by one billon matrix is *massive*. Just storing a matrix that large would take on the order of *exabytes*, which would be a notable chunk of our world's largest data centers. And even if we could somehow store the matrix, calculating the principle eigenvalue would take on the order of *decades* (it would require solving a massive linear system, which as we know requires approximately $\frac{2n^3}{3}$ flops, which is a lot even for our fastest supercomputers). The situation seems dire, but there are two saving observations:

1. The matrix A is *sparse*. Most websites only have a handful of link, especially compared to the total number of websites. This means we can use more memory efficient representations of A.

2. We're ultimately interested in determining $\lim_{k \to \infty} A^k \mathbf{u}$, but it turns out that $A^\kappa \mathbf{u}$ is a pretty good approximation for fairly small values of κ. Instead of solving for the steady-state distribution exactly, we can just multiply $\mathbf{u}$ a couple times with A. In other words, the sequence converges quite fast. This approach is called the **power method** because it consists of determining the result of multiplying a vector with a power of A by repeated multiplication by A.

We will use these two observations to write a more realistic implementation of PageRank than what is given in our textbook.

[1] In practice, we'd be looking at a subset of the internet that is relevant to the given search.

2 The Tasks

This section outlines what you will be requried to do in this lab. **Please read through these instructions carefully and entirely.**

2.1 Getting Set-up

First, I recommend reading the chapter in our textbook titled "PageRank."[2] The TFs will also cover this material during your lab section. In particular, you'll need to read about how to deal with boundaries, as well as dampening.

Next, we need to install some libraries. When doing "real world" computational linear algebra it's often best to depend on libraries written by experts. We'll be using two Python libraries in addition to SciPy and NumPy:

- **NetworkX**[3] is a package for working with and analyzing graphs. It is very fast and has a nice interface for turning graphs into matrices.
- **scikit-learn**[4] is a package for machine learning. Perhaps unsurprisingly, it has a lot of very useful functions for working with matrices. It's worth exploring, but we'll only need one function from it.

Task. You will have to install these packages in order to use them. I would recommend using pip via the following commands:

```
python3 -m pip install networkx
python3 -m pip install scikit-learn
```

You may have to replace `python3` with `python` or `py` depending your system. **Please try this early.** The course staff is willing to help you in office hours, but it will be hard to manage this the closer we get to the deadline.

2.2 Loading Graphs

We will be loading textual representations of graphs in *adjacency list format* (`.adjlist`) into our program. Each line in this format represents an edge and contains two numbers, each representing the nodes connected by the edge. A line of the form

```
i j
```

represents an edge from node `i` to `j`. So a file containing

```
0 1
1 2
2 0
```

represents a directed triangle graph (note that nodes are 0-indexed). See the NetworkX documentation for more details.[5] There is a NetworkX function called `read_adjlist` which we will use to read graphs into our program.

Task. You should take a look at the file `basic.adjlist`, which represents the graph in Figure 1: Note that **nodes are 0-indexed** in this representation so this file has values from 0 to 5. We'll be using this file for testing.

[2] https://mcrovella.github.io/CS132-Geometric-Algorithms/L19PageRank.html
[3] https://networkx.org
[4] https://scikit-learn.org/stable/
[5] https://networkx.org/documentation/stable/reference/readwrite/adjlist.html

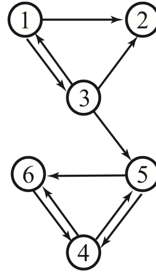

Figure 1: A basic directed graph for testing

2.3 Building Sparse Adjacency Matrices

So far in this course, we've been working with NumPy arrays to represent matrices, but NumPy arrays require too much space to represent very large matrices. We saw on a previous assignment that matrices from practical applications can sometimes be *sparse*, in that they don't have very many entries relative to their size, and this can drastically affect the performance of basic operations on them. In SciPy, there are several implementations of sparse matrices which are more space efficient and more efficient to manipulate than standard NumPy arrays. In NetworkX there is a function called `adjacency_matrix` which builds an adjacency matrix in one of these sparse implementations.

Task. Run the command `python3 pagerank.py basic.adjlist 2`. This will throw an error since you haven't implemented the power method yet, but you should see printed the adjacency list for the above graph. Make sure that it looks correct.

2.4 Normalizing Sparse Matrices

What we have at the moment is an adjacency matrix, but not one that is *stochastic*. We have to divide each column (with nonzero entries) by the *number* of nonzero entries, in order to make each column into a uniform distribution over those websites to which the current website links. This is quite a difficult thing to do efficiently, which is why there is a scikit-learn function called `normalize` for doing this.

Task. After running the same command as in the previous step, you should also see the normalized form of the adjacency matrix. Make sure that it looks correct.

2.5 One Step of the Power Method

The bulk of your work is going to be in defining the function that does one iteration of the power method. The reason this part is tricky is that dealing with boundaries and damping **make the matrix dense**, which would defeat the purpose of this using the sparse matrix in the first place.

The key observation is that both boundary-handling and damping can be implemented without changing the very large sparse normalized matrix. The equation from the textbook for the final transition matrix is

$$(1 - \alpha)A + \frac{\alpha}{n}\mathbf{1}$$

where A is the normalized matrix with all-zeros columns replaced with all-ones columns (before normalization).[6] The first step is to notice that A can be thought of as the matrix $A' + \frac{1}{n}A_z$ where A' is the normalized matrix possibly with all-zeros columns and A_z is the matrix with all-ones columns where A' has all-zeros

[6] Historically $(1 - \alpha)$ is called the *damping factor* and the original paper on PageRank takes α to be 0.15. We'll do the same.

columns and zeros everywhere else. In the context of our running example:

$$\begin{bmatrix} 0 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 1/3 & 0 & 0 & 0 \\ 1/2 & 1/6 & 0 & 0 & 0 & 0 \\ 0 & 1/6 & 0 & 0 & 1/2 & 1 \\ 0 & 1/6 & 1/3 & 1/2 & 0 & 0 \\ 0 & 1/6 & 0 & 1/2 & 1/2 & 0 \end{bmatrix}$$

is the same as

$$\begin{bmatrix} 0 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 1/3 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1/2 & 1 \\ 0 & 0 & 1/3 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{bmatrix} + \frac{1}{6} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Next, we're not actually interested in the above matrix, but the matrix-vector multiplication

$$\left((1-\alpha)A + \frac{\alpha}{n}\mathbf{1}\right)\mathbf{v}$$

for some vector $\mathbf{v}$, which can be rewritten as

$$(1-\alpha)A'\mathbf{v} + \frac{1-\alpha}{n}A_z\mathbf{v} + \frac{\alpha}{n}\mathbf{1v}$$

and the two vectors on the right in this equation have a very simple structure.

Task. You will be implementing a single step of the power method in the function `power_step`. Given a sparse matrix `A'` and a 2D NumPy array `v`, you need to implement the equation

$$(1-\alpha)A'\mathbf{v} + \frac{1-\alpha}{n}A_z\mathbf{v} + \frac{\alpha}{n}\mathbf{1v}$$

without building the matrices A_z or $\frac{\alpha}{n}\mathbf{1}$. You cannot, for example, use[7]

```
(alpha / A'.shape[0]) * np.ones(A'.shape)
```

Instead, you need to think about what the vectors $\frac{1-\alpha}{n}A_z\mathbf{v}$ and $\frac{\alpha}{n}\mathbf{1v}$ are, and compute them separately. For the first of these two, you are given a vector `zero_cols` as input which has the property that `zero_cols[i]` is 1 if the ith column of A' is all-zeros, and 0 otherwise. **This is probably the trickiest part, so give it some thought**. A couple last implementation notes:

- You should **not** use `A @ v` for matrix-vector multiplication when `A` is sparse, but rather `A.dot(v)`. This is better optimized for sparse matrices.
- Remember that adding a number to a vector adds that number entry-wise. For example,

  ```
  np.array([1, 2, 3]) + 2 * np.ones(3)
  ```

  is the same as

  ```
  np.array([1, 2, 3]) + 2.
  ```

[7]This is not a restriction of the assignment, this matrix will be too large to build on your machine.

Power Method

Now that we have pre-processed our matrix and built the function for doing an iteration of the power method, we can implement the power method itself. This is a matter of implementing the algorithm from the textbook.

Task. You will be implementing the power method in the function `power_iter`. You should separately write the function `l1_error` which computes the function

$$\text{error}_{\ell^1}(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{n} |\mathbf{u}_i - \mathbf{v}_i|$$

as in the notes. If you're interested in where the name comes from, feel free to peruse the Wikipedia page on Norms.[8]

You'll notice when you run the `pagerank.py` program a bunch of stuff is printed to the command line. In this function you **must** call the `print_error_log` once every 10 iterations of the power method, and one last time when you stop iterating (so you can see the final error). For example, a run might print the following.

```
    | error after 10 iterations: 0.03154451079478763
    | error after 20 iterations: 0.003705951856289287
    | error after 30 iterations: 0.000535150600826042
    | error after 40 iterations: 8.340084685614549e-05
    | error after 50 iterations: 1.3684642707116416e-05
    | error after 60 iterations: 2.3141829071034e-06
    | error after 70 iterations: 3.9985909018511144e-07
    | error after 80 iterations: 7.079267907127914e-08
    | error after 90 iterations: 1.2740109503149871e-08
    | error after 92 iterations: 9.043053349643915e-09
```

So all together, the run took 92 iterations. This will make it easier for you to see the progress that is being made as you run it.

2.6 Running PageRank

When you're done, you'll be able to run your program on large graphs taken from the Stanford Large Network Dataset Collection.[9] The starter code comes with three graphs based on actual Web data, one from Stanford, one from Stanford and Berkeley, and one from Google.

You will be filling in functions in the file `lab5.py`, but the actual program for PageRank is `pagerank.py`. This file imports the functions that you write.

The file `pagerank.py` takes two arguments at the command line: the name of a file in adjacency list format, and an exponent for the error tolerance used in the power method. So

```
python pagerank.py basic.adjlist 4
```

will run PageRank on the graph in `basic.adjlist` with error tolerance 0.0001. If you type

```
python pagerank.py
```

with no arguments, you will get a usage string.

The starter code also comes with a Makefile. If you haven't had a chance to work with Makefiles, they are used to organize code building, running and cleaning processes. On the command line, you can run the following commands:

- `make basic` runs PageRank on the basic graph from the running example

[8] https://en.wikipedia.org/wiki/Norm_(mathematics)#Euclidean_norm
[9] http://snap.stanford.edu/data/index.html

- `make stanford` runs PageRank on the Stanford web data
- `make berkstan` runs PageRank on the Berkeley-Stanford data
- `make google` runs PageRank on the Google data
- `make fullrun` runs PageRank on all three of the large graphs
- `make clean` deletes all auxiliary files created by the program (more on this in a moment)

The last thing you need to know about running the program: you'll find that preprocessing will take the bulk of the time when running. To make this a bit easier, the program saves intermediate representations of the pre-processed graphs and rankings it finds so that you don't have to recompute them if you rerun the program. These are the files with the extensions `.npy` and `.npz`. This has an important/unfortunate consequence: **if you find that your implementation is incorrect but you already ran it** you have to delete these files or call `make clean` to get rid of them before recomputing the ranking.

Task. You should run PageRank on the Stanford graph with an error tolerance of 10^{-8}. Alternatively, you can use `make stanford`. You should then fill in the variable `top_five_stanford` with the top 5 nodes after calling PageRank. This variable should be assigned to lists of 5 numbers. You can determine the values by looking at what is printed to the command line. (You should also do this for the other larger graphs, but you don't need to submit anything for the other graphs.)

3 Submission

All together, your required tasks are:

- Install necessary packages (Part 0)
- Verify the setup by running the program on `basic.adjlist` (Part 1)
- Fill in `power_step` (Part 4)
- Fill in `l1_error` and `power_iter` (Part 5)
- Fill in the ranking variables `top_five_stanford` (Part 6)

You are given starter code in `lab5.zip`. You'll upload a single file (found in that directory) `lab5.py` to Gradescope, where you can verify that it passes some (but not all) autograder tests. **Don't change the name of this file when you submit.** Also don't change the names of any functions included in the starter code. **The only changes you should make are to fill in the provided TODO items.** A couple last note:

- Test using `make basic` or `python3 basic.adjlist 8` as you work. This will be faster than working with the large graphs while testing. You can verify the ranking against what is given in the notes.
- This isn't a lot of code, but there's a lot to think about. Work out a couple examples on paper, make sure you know what you are trying to implement before you go to typing it out in Python.