Numerical Linear Algebra: Project 3

Page Rank implementations

Gerard Castro Castillo

December 16, 2023

C1: Compute the PR vector of M_m using the power method (adapted to PR computation). The algorithm reduces to iterate $x_{k+1} = (1-m)GDx_k + ez^t x_k$ until $\|x_{k+1} - x_k\|_\infty <$ tol.

The implementation is found at the `c1c2.py` script, which also contains the resolution of the C2 problem.

First of all, importing the corresponding routines of the `auxiliary.py` module, the link matrix G is loaded from the file and the sparse diagonal matrix D created. The latter is created computing the out-degree values n_j of a page j and, then, defining $D = \operatorname{diag}(d_{11}, \cdots, d_{nn})$, where $d_{jj} = \frac{1}{n_j}$ if $n_j \neq 0$ and $d_{jj} = 0$, otherwise.

Once defined $A = GD$ (and computed with the *built-in* `scipy` method),

the `compute_PR_with_storing` function computes *PageRank* (PR) vector (with storing) from the $M_m = (1-m)A + mS$ matrix. Essentially the algorithm iterates $x_{k+1} = M_m x_k$ until $\|x_k - x_{k+1}\|_\infty <$ `tol`, and it starts at $x_0 = (1/n, \ldots, 1/n)$ as starting point.

At most, the only non-triviality of the exercise is the $z = (z_1, \cdots, z_n)^t$ vector computation.

Since it is defined as $z_j = \begin{cases} m/n & \text{if column } j \text{ of } A \text{ contains non-zero elements,} \\ 1/n & \text{otherwise.} \end{cases}$

we need to know whether column j of matrix A contains non-zero elements. However, since A is a `scipy` (COO) sparse matrix, we can leverage the method `A.indices`. This method returns an array for each non-zero element with the index of the column where it is. Then `np.unique(A.indices)` $= [0, 2, 3, 4]$ directly retrieves the columns' indices of the matrix A with non-zero elements and, therefore, the vector z can be immediately computed.

Finally, after setting `tol` $=$ `1e-15` to resemble the machine ϵ in *Python*, the PR vector was found in around 0.18 s ($\pm$0.01 s).

C2: Compute the PR vector of M_m using the power method without storing matrices.

The implementation, also found at `c1c2.py` module, proceeds essentially the same as C1 but now using `compute_PR_without_storing` to compute the PR vector without storing the matrices (M_m, A, D, G) and from the idea provided in the statement.

In this case, the main difficulty posed is to calculate the web pages with link to page k, L_k, (and the number of outgoing links from page k, n_k), as per the step 1 (and 2) in the statement idea. However, since n_k is the length of L_k, the problem reduces to compute L_k and, to do so, the `scipy` method `indptr` for sparse (CSC) matrices can be now leveraged.

In particular, given the link matrix G (filled just with 0 or 1) and certain k, L_k is given by `G.indices[G.indptr[k]:G.indptr[k+1]]`.

This is because, the method `G.indptr` maps the elements of `data` & `indices` to rows such that, for row k, `G.indptr[i]:G.indptr[i+1]` are the `indices` of elements to take from `data` corresponding to row i.

So, if we assume `A.indptr[i]` $= j$ and `A.indptr[i+1]` $= l$, the data corresponding to row i would be at columns indices `[j: l]`, *i.e.* `A.data[j: l]`[1].

To conclude and regarding the results, the same `tol` value as before was used and the PR vector was found in around 11.3 s ($\pm$0.3 s). While the amount of *RAM* memory consumed in the calculations is much lower now, there has been a $\sim$ 100x increase in computational time. This represents the price to pay not to store any matrix.

However, as expected, the solution is "*approximately*" the same (with a difference of $1.59451 \cdot 10^{-14}$).

[1] `A.data` is an array containing all the non-zero values of the sparse matrix.