Math087 - Midterm Report 2 due 2025-03-23

George McNinch

2025-03-23

1 Instructions

There are 6 questions contained in the discussion below. You should submit to gradescope a report as a `PDF` file containing responses to those questions, written in the style of a *lab report*. You should write in such a manner that a reader who didn't know the assignment in advance could read and follow your submission.

In some cases, you are asked to write `python` code. You should include the text of the code embedded in your submission.

(And: You don't have to include any discussion on the *for-fun* section at the end. . .)

2 `PageRank` and Markov processes

Markov chains formed the original basis for the PageRank algorithm used –for example – by Google to rank web-pages by some proxy for "importance".

Consider the following basic model of surfing the web. A user begins surfing at a particular website. From the set of outgoing links from this site, a choice is made, with each link given equal probability. This transition brings the user to a new page, where the process is repeated.

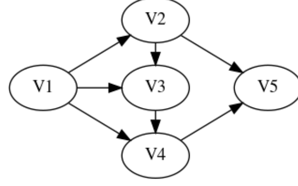
Let's model this system as a Markov Process: the nodes in our transition diagram correspond to a certain collection of websites, and there is a directed edge from node `A` to node `B` if site `A` contains a hyperlink to the site `B`. Consider a fixed node `A`: to assign the probabilities on an edge `A` $\to$ `B` we just need to count the number `N(A)` of outgoing links from site `A`; then all edges `A` $\to$ `B` will be given probability `1/N(A)`.

Number the nodes $a_0, \dots, a_{n-1}$, and for each $0 \le i \le n-1$ write $\mathbf{e}_i$ for the corresponding standard basis vector. We are going to define the *adjacency matrix* A. It should have the property that $A\mathbf{e}_i$ is the sum of all the standard basis vectors $\mathbf{e}_j$ for which there is an edge $a_i \to a_j$. A little thought shows that the s, t entry satisfies

$$A_{s,t} = \begin{cases} 1 & \text{if there is an edge } a_t \to a_s \\ 0 & \text{otherwise} \end{cases}$$

Using A, one can create the "naive" transition matrix T whose entries are the probabilities discussed above: as in the course notebooks, one views the directed graph as the transition diagram of a probabilistic finite-state machine, and T is the associated transition matrix.

Suppose the sites we consider were described by the following directed graph:



Using the procedure just outlined, we obtain the following adjacency matrix and "naive" transition matrix:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad T = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 0 & 0 \\ 1/3 & 0 & 1 & 0 & 0 \\ 0 & 1/2 & 0 & 1 & 0 \end{bmatrix}$$

Notice that `V5` is a *sink* - there are no outgoing edges (the site `V5` has no links to any of the sites under consideration). This is problematic – the transition diagram we obtain isn't `strongly connected` and the corresponding transition matrix isn't *stochastic* (the entries in the last column don't sum to 1).

Now, the model is supposed to represent a *random web-surfer*, so to recover a stochastic matrix, we model the transition from a *sink* node – i.e. a node with no "out-connections" – by having the surfer randomly choose a site (i.e. a node) from all sites with equal probability.

In the example, this corresponds to replacing the final column of T by the column

$$\frac{1}{5}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}^T$$

The matrix T_1 obtained in this manner is *stochastic*, though it may well not satisfy the conclusion of the Perron-Frobenius Theorem. For the above example, we have

$$T_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1/5 \\ 1/3 & 0 & 0 & 0 & 1/5 \\ 1/3 & 1/2 & 0 & 0 & 1/5 \\ 1/3 & 0 & 1 & 0 & 1/5 \\ 0 & 1/2 & 0 & 1 & 1/5 \end{bmatrix}$$

We now introduce a *damping factor*:

> The PageRank theory holds that an imaginary surfer who is randomly clicking on links will eventually stop clicking. The probability, at any step, that the person will stop is a damping factor p.

We model this stipulation by declaring that with probability `p` the imaginary surfer will "stop clicking". The transition that occurs in this case lands the surfer on a new node chosen at random from *all* available nodes (with equal probability).

For the example, for the above diagram, we obtain the matrix

$$C = (1-p)T_1 + \frac{p}{5}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

which we call the `PageRank` transition matrix for the damping probability p.

3 Questions

3.1 Describing the transition matrix

Given a adjacency matrix A, complete the following description of the `i,j` entry of the corresponding `PageRank` transition matrix C for the probability p.

- If the `j`-th column of the matrix A is equal to $\mathbf{0}$, then for each `i=0,...,n-1`

  $C_{i,j} =$ __________________________

- Suppose that the sum of the entries of the `j`-th column of the matrix A is equal to $s > 0$, then for each `i=0,...,n-1`:

  If $A_{i,j} = 0$, then $C_{i,j} =$ __________________________

  If $A_{i,j} = 1$, then $C_{i,j} =$ __________________________

3.2 Coding the transition matrix

Use your description in 1. to write a `python` function to create the `PageRank` transition matrix from an adjacency matrix A.

Here is a suggested starting point:

```
import numpy as np

def make_transition(A,p):
    (n,m) = A.shape
    if n==m:                ## A must be a square matrix

        C = np.zeros((n,n))   ## create a matrix of
        #                     ## zeros of the correct size
        #
        #                     ## insert code to modify
        #                     ## the entries C[i,j]
        #
        return C
```

Your responses to Section 3.1 above should be helpful in producing the required code.

- sums

  Recall that you can obtain the *sum* `s` of the entries of the `i` th column of the matrix `A` using the following code:

  ```
  s=sum(A[:,i])
  ```

- testing for a zero column

  And you can *test* whether the `i~th column of ~A` is equal to the zero-vector using

```
(A[:,i] == 0).all()
```

This expression will be `True` if all entries in the `i` th column are equal to 0.

You can use this expression in a conditional statement:

```
if (A[:,i] == 0).all() then
    ## do something...
```

- looping

  I recommend that you *build up* the matrix `C` by assigning values to the entries one column at a time, by using a *loop* as follows:

```
for j in range(n):
    ## insert code to assign values for the j-th column...
```

- You can assign the value of `C` in the `i` th row and `j` th column using an expression like

```
C[i,j] = ...
```

3.3 Testing your code

Test the code you produced in Section 3.2 on the example from Section 2 by proceeding as follows:

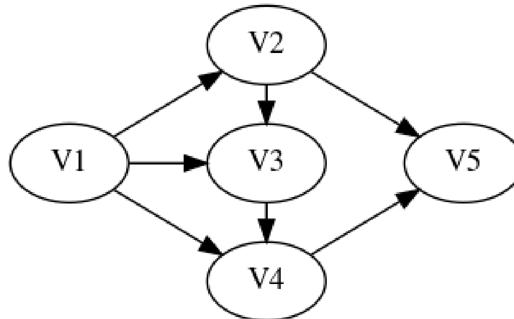
```
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})
A = np.array([[ 0, 0, 0, 0, 0 ],
              [ 1, 0, 0, 0, 0 ],
              [ 1, 1, 0, 0, 0 ],
              [ 1, 0, 1, 0, 0 ],
              [ 0, 1, 0, 1, 0 ]])

## the following should result in the output indicated just below:
make_transition(A,p=0.8)
```

```
array([[0.16000, 0.16000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.16000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.26000, 0.16000, 0.16000, 0.20000],
       [0.22667, 0.16000, 0.36000, 0.16000, 0.20000],
       [0.16000, 0.26000, 0.16000, 0.36000, 0.20000]])
```

3.4 Find the example page ranking using your code

Find a 1-eigenvector for the `PageRank` transition matrix `C` when `p=.8` and when `p=.4` for the directed graph



Be sure to normalize your eigenvector to obtain a *probability vector*.

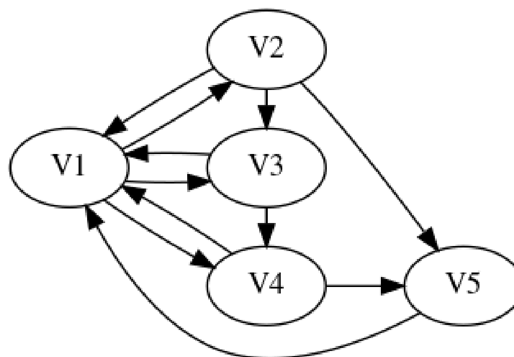
Explain what the entries in this vector predict about the probability of a random surfer landing on one of the five web-sites corresponding to the nodes in the diagram. What is the probability that a random surfer arrives at the top-rated site on a given click?

Compare the information you get from the eigenvector with that obtained by studying powers of `C` calculated using `np.linalg.matrix_power(C,m)`.

Remark: `PageRank` uses the `long-term probability` as a proxy for the importance of the page – the probability with which a random surfer lands on a page in the long-run determines its ranking.

3.5 `PageRank` for another example

The following diagram describes the same nodes as in problem 2., but includes some additional edges (i.e. links).



Assess the impact of the additional edges on the page rankings (when `p=.8` and when `p=.4`) – i.e. compare the ranking obtained for this diagram with that obtained in problem 2. Before making your comparison, you'll need to first find the adjacency matrix for this new configuration, and then use your code to find the corresponding `PageRank` transition matrix.

3.6 `PageRank` transition matrices and the Frobenius-Perron Theorem

For *any* directed graph, explain why the corresponding `PageRank` transition matrix for `p>0` is a stochastic matrix corresponding to a strongly connected aperiodic transition diagram. In particular, explain why the conclusion of the *Perron-Frobenius Theorem* holds for this matrix.

3.7 A larger example

We are going to consider some data which is stored in JSON format. This format is quite similar to that of a `python` dictionary. The role of the sites in our web-surfing model will be played by animal names, and a link is then just a "from"-"to" pair; for example, here are two such pairs:

```
[
  {
    "from": "Blue Whale",
    "to": "Snail"
  },
  {
    "from": "Blue Whale",
    "to": "Alligator"
  }
]
```

So for example, the first link indicates that the "Blue Whale" page has a link to the "Snail" page.

Data is provided to you in the form of a `JSON` file. The code below takes as an argument the name of the `JSON` file and returns a pair `(sites,A)` where `sites` is a list of strings containing the name of each "site" that appears in one of the `from/to` pairs, and `A` is the adjacency matrix determined by the indicated links.

```
import json
import numpy as np
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})

def bv(it,items):
    return np.array([1.0 if i == items.index(it)
                     else 0.0 for i in range(len(items))])

## >>> bv("c",["a","b","c","d"])
## array([0.00000, 0.00000, 1.00000, 0.00000])

def adj_from_json(json_file):
  # extract the list of sites and the adjacency matrix from the JSON
  # file named `json_file`
  with open(json_file) as f:
      adj_data = json.load(f)

  dict = {}

  for i in adj_data:
      lfrom = i['from']
      lto   = i['to']
      if lfrom in dict.keys():
          dict[lfrom].add(lto)
```

```
    else:
        dict[lfrom] = set()
        dict[lfrom].add(lto)
    if not(lto in dict.keys()):
        dict[lto] = set()

sites = list(dict.keys())

A = np.array([sum([bv(l_to,sites) for l_to in dict[l_from]],
                  np.zeros(len(sites)))
              for l_from in sites])
return (sites, A)
```

(If you are curious how it works, the code

```
with open(json_file) as f:
    adj_data = json.load(f)
```

transforms the data in the indicated `json_file` into a list of `python` dictionaries.)

Now, here is a file containing data from which you should build an adjacency matrix using the function `adj_from_json`: data file See Section 4 below for some discussion of how one should use this file.

Once you have produced the adjacency matrix `A` from `data.json`, use the `make_transition` function you wrote in Section 3.2 to build the `PageRank` transition matrix `C` from the adjacency matrix `A`, for `p=0.8`

Now find the page-rankings of these `pages`, using both the eigenvector method and the power-iteration method. For each method, report the top ten "sites" by name (when `p=0.8`). When using the power-iteration method, report the number of iterations used.

See Section 5 below for some discussion of how you might extract the top ten sites.

> Note that you can get the site name corresponding to index `i` as folows. If you used `(ll,A) = adj_from_json("data.json")` to build the adjacency matrix then the list `ll` contains the names, and in particular `ll[i]` is the name for index `i`.
>
> Conversely,
>
> ```
> ll.index('Horse')
> ```
>
> returns the index `i` for which `ll[i] == 'Horse'`.

What happens if you reduce the value of `p`? How much must you vary `p` to see any change in the "top-ten" list?

3.8 Other applications of `PageRank`

The damping probability $0 < p \leq 1$ forces the `PageRank` transition matrix to satisfy the conclusion of the Perron-Frobenius Theorem. As we discussed above, from the point-of-view of ranking web pages this quantity represents the probability that a random web-surfer gets bored and makes a random new choice of sites. In particular, this provides a reasonable *self-contained* explanation for the use of this `PageRank` transition matrix.

On the other hand, one might hope to use the PageRank approach to rank other "linked material". For example, one might try to rank published academic papers by citation.

More precisely: given a collection of academic papers, consider a directed graph whose nodes are the papers and for which there is a directed edge from *paper A* to *paper B* if *paper A* contains a citation to *paper B*.

Suppose that each paper in the collection has at least one citation to another paper in the collection; under this assumption, the "naive" transition matrix T *is* stochastic. But this transition matrix may fail to satisfy the conclusion of the Perron-Frobenius Theorem. One can "fix" this problem as before by introducing a "damping probability" $0 < p \le 1$ i.e. by replacing T with the matrix

$$(1-p)T + \frac{p}{N}\mathbf{1}^{N\times N}$$

where N is the number of papers in the collection and $\mathbf{1}^{N\times N}$ denotes the $N \times N$ matrix all of whose entries are equal to 1.

Do you think this is a reasonable proposal? What would be the explanation for the damping probability from the point-of-view of the model?

4 Appendix: instructions for using the JSON file

In Section 3.7, you are asked to use JSON data found in a file that you must download. To use this file, follow these instructions:

- Local computer instructions

  If you use python on your local computer, save this file to your drive and run python from a shell in the same directory. You can then call

  ```
  (ll,A) = adj_from_json("data.json")
  ```

  to extract the required information from this file. After executing this code, ll is the list of sites specified in the file, and A is the adjacency matrix.

- Colab instructions

  If you use colab, proceed as follows. Save the file data.json to your local computer, and then upload that file to your google drive.

  To enable colab to see your google drive, execute the following commands in a cell:

  ```
  from google.colab import drive
  drive.mount('/gdrive')
  ```

  Now change to the correct directory in your google drive by executing the following in a cell in colab (you really do need the % symbol):

  ```
  %cd /gdrive/My\ Drive/
  ```

  Note for example that you can use the following command to see the contents of the currently selected directory of your google drive:

```
%ls
```

Now you can run

```
(ll,A) = adj_from_json("data.json"):
```

in a `colab cell`. The file named `data.json` will be read from your google drive. After executing this code, `ll` is the list of sites specified in the file, and `A` is the adjacency matrix.

5 Appendix: sorting data

In carrying out Section 3.7 you will have found a list of probabilities which correspond to *web pages*. You then need to find the *top 10 pages*. The following example will show you some techniques to use to do this in *python*.

If you have a list in python, you'd like to be able to extract (say) the top 10 elements of this list.

This is reasonably straightforward. Let me show how. I'll use `numpy` to generate a (random) list of 20 numbers:

```
import numpy as np

L = np.random.rand(20)
L
```

```
array([0.12039294, 0.78665645, 0.36627182, 0.53842076, 0.57620245,
       0.034068  , 0.00856583, 0.94600437, 0.33425539, 0.28685265,
       0.79008222, 0.53609803, 0.16978149, 0.64130782, 0.29008797,
       0.25703727, 0.97788573, 0.69052141, 0.01364596, 0.4754929 ])
```

To extract (say) the top 5, I can **sort** the list.

```
L.sort()
L
```

```
array([0.00856583, 0.01364596, 0.034068  , 0.12039294, 0.16978149,
       0.25703727, 0.28685265, 0.29008797, 0.33425539, 0.36627182,
       0.4754929 , 0.53609803, 0.53842076, 0.57620245, 0.64130782,
       0.69052141, 0.78665645, 0.79008222, 0.94600437, 0.97788573])
```

This gives the numbers in low-to-high order. We can reverse this if we like:

```
np.flip(L)
```

```
array([0.97788573, 0.94600437, 0.79008222, 0.78665645, 0.69052141,
       0.64130782, 0.57620245, 0.53842076, 0.53609803, 0.4754929 ,
       0.36627182, 0.33425539, 0.29008797, 0.28685265, 0.25703727,
       0.16978149, 0.12039294, 0.034068  , 0.01364596, 0.00856583])
```

And then easily extract the top 5:

```
np.flip(L)[0:5]
```

```
array([0.97788573, 0.94600437, 0.79008222, 0.78665645, 0.69052141])
```

However, this leaves us with a problem. After sorting, we no longer know the **original positions** of these top-five values.

Let's Consider a list of 20 identifiers (I've scraped up some *color names* to use as identifiers), and let's generate a new list of 20 numbers paired with these identifiers

```
rng = np.random.default_rng()
import pprint
float_formatter = "{:.5f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})

colors = [ "Gainsboro",
           "Gamboge",
           "Glossy grape",
           "Gold (metallic)",
           "Gold (Crayola)",
           "Golden poppy",
           "Golden yellow",
           "Goldenrod",
           "Gotham green",
           "Granite gray",
           "Granny Smith apple",
           "Gray (web)",
           "Gray (X11 gray)",
           "Green",
           "Green (Crayola)",
           "Green (web)",
           "Green (Munsell)",
           "Green (pigment)",
           "Green-blue",
           "Green Lizard"
           ]

vals = rng.random(20)

for (c,v) in zip(colors,vals):
    print(f" {c:20} {v}")
```

```
Gainsboro            0.8775233646973506
Gamboge              0.8186445313681252
Glossy grape         0.5340385053741519
Gold (metallic)      0.5532655442083644
```

```
Gold (Crayola)        0.5685351328046194
Golden poppy          0.24123175028452726
Golden yellow         0.38647426037794086
Goldenrod             0.5485217960583766
Gotham green          0.11040288793210606
Granite gray          0.6150425887527425
Granny Smith apple    0.7440640331391108
Gray (web)            0.629503395347427
Gray (X11 gray)       0.06006192199989513
Green                 0.5663684687964026
Green (Crayola)       0.05591663061129326
Green (web)           0.2329656880717743
Green (Munsell)       0.1122235620433173
Green (pigment)       0.4627926185666442
Green-blue            0.4119034460203417
Green Lizard          0.417922877583509
```

Let's assume that the `i` th entry of the array `val` corresponds to the *popularity* of the `i` th entry of `colors`. Suppose we want to know the top 5 most popular colors, in order.

The `numpy` function `argsort` provides a way to keep track of the original indices of the sorted elements. More precisely,

```
np.argsort(vals)
```

```
array([14, 12,  8, 16, 15,  5,  6, 18, 19, 17,  2,  7,  3, 13,  4,  9, 11,
       10,  1,  0])
```

The output of `argsort` is a list of *indices*; it returns an array of indices of the same shape as its input that index the data in sorted order.

We can obtain the sorted list (in ascending order) of values as follows:

```
np.array([ vals[i] for i in np.argsort(vals) ])
```

```
array([0.05592, 0.06006, 0.11040, 0.11222, 0.23297, 0.24123, 0.38647,
       0.41190, 0.41792, 0.46279, 0.53404, 0.54852, 0.55327, 0.56637,
       0.56854, 0.61504, 0.62950, 0.74406, 0.81864, 0.87752])
```

And we can get the sorted list of colors (in ascending order of the corresponding value):

```
np.array([ colors[i] for i in np.argsort(vals) ])
```

```
array(['Green (Crayola)', 'Gray (X11 gray)', 'Gotham green',
       'Green (Munsell)', 'Green (web)', 'Golden poppy', 'Golden yellow',
       'Green-blue', 'Green Lizard', 'Green (pigment)', 'Glossy grape',
       'Goldenrod', 'Gold (metallic)', 'Green', 'Gold (Crayola)',
       'Granite gray', 'Gray (web)', 'Granny Smith apple', 'Gamboge',
       'Gainsboro'], dtype='<U18')
```

If we want *descending* order, we can ask for the `argsort` of the additive inverse of the vector. For example

```
np.array([ vals[i] for i in np.argsort( -vals) ])
```

```
array([0.87752, 0.81864, 0.74406, 0.62950, 0.61504, 0.56854, 0.56637,
       0.55327, 0.54852, 0.53404, 0.46279, 0.41792, 0.41190, 0.38647,
       0.24123, 0.23297, 0.11222, 0.11040, 0.06006, 0.05592])
```

```
np.array([ colors[i] for i in np.argsort( -vals) ])
```

```
array(['Gainsboro', 'Gamboge', 'Granny Smith apple', 'Gray (web)',
       'Granite gray', 'Gold (Crayola)', 'Green', 'Gold (metallic)',
       'Goldenrod', 'Glossy grape', 'Green (pigment)', 'Green Lizard',
       'Green-blue', 'Golden yellow', 'Golden poppy', 'Green (web)',
       'Green (Munsell)', 'Gotham green', 'Gray (X11 gray)',
       'Green (Crayola)'], dtype='<U18')
```

And we could have obtained the top three most popular colors together with their rankings via:

```
pop = np.array([ (colors[i],vals[i]) for i in np.argsort( -vals) ])
pop[:3]
```

```
array([['Gainsboro', '0.8775233646973506'],
       ['Gamboge', '0.8186445313681252'],
       ['Granny Smith apple', '0.7440640331391108']], dtype='<U32')
```