COMPUTER SCIENCE AND ENGINEERING
Indian Institute of Technology, Palakkad
CS5016: Computational Methods and Applications

28 Mar, 2024

Coding Assignment 9

Eigen and Singular-value Decomposition

1. Create a class `Matrix` that stores the matrix as an `numpy.array`. This class should enable the following

   - Initialize an $m \times n$ zero matrix.

```
m = Matrix(3, 4)
```

   - Get and set values in the matrix

```
m = Matrix(3, 4)
m[1, 1] = 2; print(m[2, 3])
```

   - Print the matrix; each element is displayed with scale of 8.

```
m = Matrix(3, 4)
m[1, 1] = 2; m[2, 3] = 4; print(m)
```

   Expected output:

```
A 3 x 4 matrix with entries:
0.00000000      0.00000000      0.00000000      0.00000000
0.00000000      2.00000000      0.00000000      0.00000000
0.00000000      0.00000000      0.00000000      4.00000000
```

   - Convert to a diagonal matrix.

```
m = Matrix(3, 4); m.toEye(); print(m)
m = Matrix(4, 3); m.toEye(); print(m)
```

   Expected output:

```
A 3 x 4 matrix with entries:
1.00000000      0.00000000      0.00000000      0.00000000
0.00000000      1.00000000      0.00000000      0.00000000
0.00000000      0.00000000      1.00000000      0.00000000

A 4 x 3 matrix with entries:
1.00000000      0.00000000      0.00000000
0.00000000      1.00000000      0.00000000
0.00000000      0.00000000      1.00000000
0.00000000      0.00000000      0.00000000
```

- Convert to a matrix of all ones.

```
m = Matrix(3, 4); m.toOne(); print(m)
```

Expected output:

```
A 3 x 4 matrix with entries:
1.00000000      1.00000000      1.00000000      1.00000000
1.00000000      1.00000000      1.00000000      1.00000000
1.00000000      1.00000000      1.00000000      1.00000000
```

- Sample a random matrix where each element is uniformly sampled from the interval $[0, 1]$.

```
m = Matrix(2, 3); m.randomize(); print(m)
```

Expected output:

```
A 2 x 3 matrix with entries:
0.94376475      0.92571413      0.03075120
0.03602654      0.52683039      0.46335698
```

- Obtain the transpose.

```
m = Matrix(2, 3); m.randomize()
print(m); print(m.t())
```

Expected output:

```
A 2 x 3 matrix with entries:
0.51879979      0.31696067      0.40351422
0.78846218      0.32580613      0.03314742

A 3 x 2 matrix with entries:
0.51879979      0.78846218
0.31696067      0.32580613
0.40351422      0.03314742
```

- Add, subtract and multiply with other `Matrix` objects. *Note: Dimension incompatibility should raise exception.*

```
m1 = Matrix(2, 3); m1.randomize(); print(m1)
m2 = Matrix(3, 2); m2.randomize(); print(m2)
print(m1 + m2.t()); print(m2.t() - m1); print(m1 * m2); print(m1 * m2.t())
```

Expected output:

```
A 2 x 3 matrix with entries:
0.74328361      0.84469456      0.48365550
0.77458901      0.91798099      0.98196750
```

```
A 3 x 2 matrix with entries:
0.29570858      0.39492569
0.48213723      0.93218397
0.04782401      0.38696604

A 2 x 3 matrix with entries:
1.03899219      1.32683179      0.53147952
1.16951469      1.85016495      1.36893354

A 2 x 3 matrix with entries:
-0.44757503     -0.36255733     -0.43583149
-0.37966332      0.01420298     -0.59500145

A 2 x 2 matrix with entries:
0.65018438      1.26811077
0.71860705      1.54162033

Exception: Matrix dimension mismatch
```

- Left and right multiplication by a scalar.

```
m = Matrix(2, 3); m.randomize(); print(m)
m = 2.0 * m; print(m);
m = m * 3.0; print(m)
```

Expected output:

```
A 2 x 3 matrix with entries:
0.39717121      0.05920897      0.04560939
0.89878582      0.07410581      0.60408590

A 2 x 3 matrix with entries:
0.79434241      0.11841794      0.09121877
1.79757163      0.14821162      1.20817181

A 2 x 3 matrix with entries:
2.38302723      0.35525383      0.27365632
5.39271489      0.44463486      3.62451543
```

- Obtain $L_{p,q}$-norms, i.e., $\|\boldsymbol{A}\|_{p,q} = \left(\sum_{j=1}^{n}\left(\sum_{i=1}^{m}|a_{ij}|^p\right)^{q/p}\right)^{1/q}$

```
m = Matrix(2, 3); m.randomize(); print(m)
print(m.norm(1))
print(m.norm(2))
print(m.norm(math.inf))
```

Question 1 continues on the next page...

Expected output:

```
A 2 x 3 matrix with entries:
0.69907995      0.74917486      0.91068809
0.94984428      0.65229157      0.40772897

4.3688077167618715
1.8366436512256366
0.9498442801300863
```

- Find a non-trivial vector $\boldsymbol{x}$ such that $\boldsymbol{Ax} = \boldsymbol{0}$, if one exists. *Hint: Use* `numpy.linalg.qr` *to the find the QR decomposition of* $\boldsymbol{A}$*. Then use backward substitution method on the upper triangular matrix.*

```python
m = Matrix(2, 3)
for i in range(2):
 for j in range(3):
  m[i, j] = (i+1) + (j+1)
print(m); print(m.solvezero())
```

Expected output:

```
A 2 x 3 matrix with entries:
2.00000000      3.00000000      4.00000000
3.00000000      4.00000000      5.00000000

A 3 x 1 matrix with entries:
1.00000000
-2.00000000
1.00000000
```

```python
m = Matrix(3, 3)
for i in range(3):
 for j in range(3):
  m[i, j] = (i+1) ** (j+1)
print(m); print(m.solvezero())
```

Expected output:

```
A 3 x 3 matrix with entries:
1.00000000      1.00000000      1.00000000
2.00000000      4.00000000      8.00000000
3.00000000      9.00000000      27.00000000

A 3 x 1 matrix with entries:
0.00000000
-0.00000000
-0.00000000
```

Continued on next page...

- Obtain the dominant Eigenvalue and the corresponding normalized Eigenvector (2-norm) using the power method.

```
m = Matrix(3, 3)
for i in range(3):
 for j in range(3):
  m[i, j] = (i+1) ** (j+1)
print(m)
e, v = m.dominantEigen()
print(e); print(v)
```

Expected output:

```
A 3 x 3 matrix with entries:
1.00000000      1.00000000      1.00000000
2.00000000      4.00000000      8.00000000
3.00000000      9.00000000      27.00000000

29.942767500676958
A 3 x 1 matrix with entries:
0.04323031
0.29744473
0.95375981
```

2. Let $\boldsymbol{A}$ be an $n \times n$ real symmetric matrix with (distinct) non-zero eigenvalues

$$|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n| > 0$$

and normalized (2-norm) eigenvectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$. Consider the following 'deflated' matrix

$$\boldsymbol{B} = \boldsymbol{A} - \lambda_1 \boldsymbol{v}_1 \boldsymbol{v}_1^T$$

It is well-know that eigenvectors corresponding to distinct eigenvalues of a symmetric matrix are orthogonal. Consequently, eigenvalues of matrix $\boldsymbol{B}$ are $0, \lambda_2, \ldots, \lambda_n$ with eigenvectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n$. Now, an application of power method on matrix $\boldsymbol{B}$ would give us the second largest eigenvalue λ_2 and its corresponding eigenvector $\boldsymbol{v}_2$. Add a function `deflate` to the `Matrix` class that uses the deflation technique to obtain all eigenvalues and corresponding normalized eigenvectors (2-norm) of a real symmetric matrix. Compare the results with that obtained using `numpy.linalg.eig`. Read about the drawbacks of deflation technique. [20]

```
m = Matrix(4, 4)
m.randomize()
m = m + m.t()
e, v = m.deflate()
print(e)
print(v) # v[:,i] is the eigenvector for eigenvalue e[i,0]
print(numpy.linalg.eig(m.mat))
```

Continued on next page. . .

Expected output:

```
A 4 x 1 matrix with entries:
5.08518615
0.51677768
0.64554130
-0.85542233

A 4 x 4 matrix with entries:
0.46798335      0.40222720      0.00000000      0.00000000
0.39766744      0.00000000      0.36459816      0.00000000
0.57253289      0.91553988      0.00000000      0.71755578
0.54319268      0.00000000      0.93116496      0.69650104

(array([ 5.08518615,  1.22938231, -0.49834511, -0.97401628]),
array([[ 0.4679833 ,  0.27047021, -0.67260455, -0.50541133],
       [ 0.39766739, -0.25605444,  0.63370162, -0.61214298],
       [ 0.57253292,  0.61563841,  0.31494746,  0.44045828],
       [ 0.54319272, -0.69445758, -0.21641041,  0.41932907]]))
```

3. Enhance the `deflate` function so that it works for $n \times n$ real symmetric matrices with repeated and zero eigenvalues. *Hint: cleverly pick the initial vector in the power iteration phase and use the fact that an $n \times n$ matrix has n eigenvalues.* [10]

```
m = Matrix(5, 5)
m[0,0] = 2; m[1,1] = 3; m[2,2] = 2
print(m)
e, v = m.deflate()
print(e)
print(v) # v[:,i] is the eigenvector for eigenvalue e[i,0]
print(numpy.linalg.eig(m.mat))
```

Expected output:

```
A 5 x 5 matrix with entries:
2.00000000      0.00000000      0.00000000      0.00000000      0.00000000
0.00000000      3.00000000      0.00000000      0.00000000      0.00000000
0.00000000      0.00000000      2.00000000      0.00000000      0.00000000
0.00000000      0.00000000      0.00000000      0.00000000      0.00000000
0.00000000      0.00000000      0.00000000      0.00000000      0.00000000

A 5 x 1 matrix with entries:
3.00000000
2.00000000
0.00000000
0.00000000
```

Continued on next page. . .

```
2.00000000

A 5 x 5 matrix with entries:
0.00000000      0.00000000      0.00000000      0.00000000      1.00000000
1.00000000      0.00000000      0.00000000      0.00000000      0.00000000
0.00000000      1.00000000      0.70710678      0.00000000      0.00000000
0.00000000      0.00000000      0.70710678      0.70710678      0.00000000
0.00000000      0.00000000      0.00000000      0.70710678      0.00000000

(array([2., 3., 2., 0., 0.]),
array([[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.]]))
```

4. Add a function `qreig` to the `Matrix` class that uses the un-shifted QR algorithm to obtain all eigenvalues and corresponding normalized eigenvectors (2-norm) of a real symmetric matrix. How does the run-time of this method compare to the `deflate` method? *Hint: Use QR algorithm to get eigenvalues. Then, obtain eigenvector* $\boldsymbol{x}$ *for* λ *by solving* $(\boldsymbol{A} - \lambda \boldsymbol{I})\boldsymbol{x} = \boldsymbol{0}$. [30]

```
m = Matrix(4, 4);  m.randomize(); m = m + m.t()
e, v = m.qreig()
print(e)
print(v) # v[:,i] is the eigenvector for eigenvalue e[i,0]
print(numpy.linalg.eig(m.mat))
```

Expected output:

```
A 4 x 1 matrix with entries:
4.25345216
-0.88261918
-0.68994490
0.60677126

A 4 x 4 matrix with entries:
0.45520947      0.82390391     -0.32588869     -0.08810930
0.56494298     -0.42892699     -0.10690409     -0.69672992
0.56078923     -0.34046789     -0.26817218      0.70546491
0.39892256      0.14589588      0.90025094      0.09551665

(array([ 4.25345216,  0.60677126, -0.88261918, -0.6899449 ]),
array([[ 0.45520947,  0.0881093 ,  0.82390391, -0.32588869],
       [ 0.56494298,  0.69672992, -0.42892699, -0.10690409],
       [ 0.56078923, -0.70546491, -0.34046789, -0.26817218],
       [ 0.39892256, -0.09551665,  0.14589588,  0.90025094]]))
```

Continued on next page...

5. Add a function `svd` to the `Matrix` class that computes the full SVD decomposition of a real matrix. *Note: You are supposed to use the approach discussed in class.* [20]

```
m = Matrix(3, 4); m.randomize(); u, s, v = m.svd()
print(u * u.t())
print(s)
print(v * v.t())
print(m)
print(u * s * v.t())
```

Expected output:

```
A 3 x 3 matrix with entries:
1.00000000     -0.00000000      0.00000000
-0.00000000      1.00000000     -0.00000000
0.00000000     -0.00000000      1.00000000

A 3 x 4 matrix with entries:
1.56552914      0.00000000      0.00000000      0.00000000
0.00000000      0.73221592      0.00000000      0.00000000
0.00000000      0.00000000      0.41299194      0.00000000

A 4 x 4 matrix with entries:
1.00000000     -0.00000000     -0.00000000     -0.00000000
-0.00000000      1.00000000      0.00000000      0.00000000
-0.00000000      0.00000000      1.00000000     -0.00000000
-0.00000000      0.00000000     -0.00000000      1.00000000

A 3 x 4 matrix with entries:
0.03247691      0.48627665      0.25196672      0.80268772
0.05345573      0.87158564      0.74148352      0.48550559
0.59317376      0.00237811      0.55125945      0.09213673

A 3 x 4 matrix with entries:
0.03247691      0.48627665      0.25196672      0.80268772
0.05345573      0.87158564      0.74148352      0.48550559
0.59317376      0.00237811      0.55125945      0.09213673
```

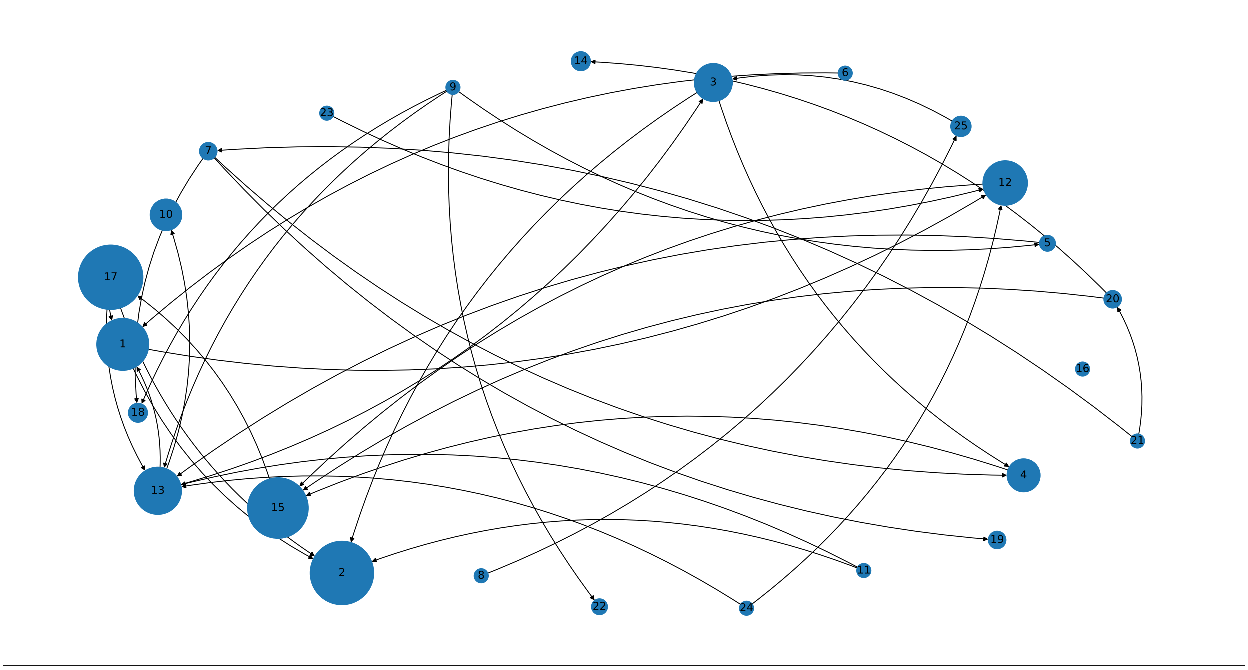
6. Modify the class `UndirectedGraph` to obtain the class `DirctedGraph` that represented a graph with directed edges. To this class add a function `pagerank` that computes and visualizes the pagerank (with damping factor) of its node. *Note: Visit `https://en.wikipedia.org/wiki/PageRank` to know about computing pagerank with damping factor. You are expected to use `numpy.linalg.eig` to obtain the page rank.* [20]

```
p = 0.05; n = 25
g = DirectedGraph(n)
```

Question 6 continues on the next page. . .

```
for i in range(n):
 for j in range(n):
  if i != j and random.random() <= p:
          g.addEdge(i+1, j+1)
g.pagerank(0.85) # function argument is the damping factor
```

Expected output:



*Note: Use the **networkx** package to to visualize the above graph. In the above representation of the graph g, size of nodes is proportional to its pagerank.*