1 The Drazin Inverse

Lab Objective: *The Drazin inverse of a matrix is a pseudoinverse which preserves certain spectral properties of the matrix. In this lab we compute the Drazin inverse using the Schur decomposition, then use it to compute the effective resistance of a graph and perform link prediction.*

Definition of the Drazin Inverse

The *index* of an $n \times n$ matrix A is the smallest nonnegative integer k for which $\mathscr{N}(A^k) = \mathscr{N}(A^{k+1})$. The *Drazin inverse* A^D of A is the unique $n \times n$ matrix satisfying the following properties.

- $AA^D = A^DA$
- $A^{k+1}A^D = A^k$
- $A^DAA^D = A^D$

Note that if A is *invertible*, in which case $k = 0$, then $A^D = A^{-1}$. On the other hand, if A is *nilpotent*, meaning $A^j = \mathbf{0}$ for some nonnegative integer j, then A^D is the zero matrix.

Problem 1. Write a function that accepts an $n \times n$ matrix A, the index k of A, and an $n \times n$ matrix A^D. Use the criteria described above to determine whether or not A^D is the Drazin inverse of A. Return `True` if A^D satisfies all three conditions; otherwise, return `False`.

Use the following matrices as test cases for your function.

$$A = \begin{bmatrix} 1 & 3 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad A^D = \begin{bmatrix} 1 & -3 & 9 & 81 \\ 0 & 1 & -3 & -18 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad k = 1$$

$$B = \begin{bmatrix} 1 & 1 & 3 \\ 5 & 2 & 6 \\ -2 & -1 & -3 \end{bmatrix}, \quad B^D = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad k = 3$$

(Hint: `np.allclose()` and `np.linalg.matrix_power()` may be useful).

Computing the Drazin Inverse

The Drazin inverse is often defined theoretically in terms of the eigenprojections of a matrix. However, eigenprojections are often costly or unstable to calculate, so we resort to a different method to calculate the Drazin inverse.

Every $n \times n$ matrix A can be written in the form

$$A = S^{-1} \begin{bmatrix} M & \mathbf{0} \\ \mathbf{0} & N \end{bmatrix} S, \tag{1.1}$$

where S is a change of basis matrix, M is nonsingular, and N is nilpotent. Then the Drazin inverse can be calculated as

$$A^D = S^{-1} \begin{bmatrix} M^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix} S. \tag{1.2}$$

To put A into the form in (1.1), we can use the *Schur decomposition* of A, given by

$$A = QTQ^{-1}, \tag{1.3}$$

where Q is orthonormal and T is upper triangular. Since T is similar to A, the eigenvalues of A are listed along the diagonal of T. If A is singular, at least one diagonal entry of T must be 0.

In general, Schur decompositions are not unique; the eigenvalues along the diagonal of T can be reordered. To find M, N, and S, we compute the Schur decomposition of A twice, ordering the eigenvalues differently in each decomposition.

First, we sort so that the nonzero eigenvalues are listed first along the diagonal of T. Then, if k is the number of nonzero eigenvalues, the upper left $k \times k$ block of T forms the nonsingular matrix M, and the first k columns of Q form the first k columns of the change of basis matrix S.

Computing the decomposition a second time, we reorder so that the 0 eigenvalues are listed first along the diagonal of T. Then the upper left $(n-k) \times (n-k)$ block forms the nilpotent matrix N, and the first $n-k$ columns of Q form the last $n-k$ columns of S. This completes a change of basis matrix that will put A into the desired block diagonal form. Lastly, we use (1.2) to compute A^D.

SciPy's `la.schur()` is a routine for computing the Schur decomposition of a matrix, but it does not automatically sort it by eigenvalue. However, sorting can be accomplished by specifying the `sort` keyword argument. Given an eigenvalue, the sorting function should return a boolean indicating whether to sort that eigenvalue to the top left of the diagonal of T.

```
>>> from scipy import linalg as la

# The standard Schur decomposition.
>>> A = np.array([[0,0,2],[-3,2,6],[0,0,1]])
>>> T,Z = la.schur(A)
>>> T                       # The eigenvalues (2, 0, and 1) are not sorted.
array([[ 2., -3.,  6.],
       [ 0.,  0.,  2.],
       [ 0.,  0.,  1.]])

# Specify a sorting function to get the desired result.
>>> f = lambda x: abs(x) > 0
>>> T1,Z1,k = la.schur(A, sort=f)
>>> T1
```

```
array([[ 2.        ,  0.        ,  6.70820393],
       [ 0.        ,  1.        ,  2.        ],
       [ 0.        ,  0.        ,  0.        ]])
>>> k                           # k is the number of columns satisfying the sort,
2                               # which is the number of nonzero eigenvalues.
```

The procedure for finding the Drazin inverse using the Schur decomposition is given in Algorithm 1.1. Due to possible floating point arithmetic errors, consider all eigenvalues smaller than a certain tolerance to be 0.

Algorithm 1.1

1: **procedure** DRAZIN(A, tol)
2: $(n, n) \leftarrow \text{shape}(A)$
3: $T_1, Q_1, k_1 \leftarrow \text{schur}(A, |x| > \text{tol})$ ▷ Sort the Schur decomposition with 0 eigenvalues last.
4: $T_2, Q_2, k_2 \leftarrow \text{schur}(A, |x| \leq \text{tol})$ ▷ Sort the Schur decomposition with 0 eigenvalues first.
5: $U \leftarrow [Q_{1_{:,:k_1}} \mid Q_{2_{:,:n-k_1}}]$ ▷ Create change of basis matrix.
6: $U^{-1} \leftarrow \text{inverse(U)}$
7: $V \leftarrow U^{-1}AU$ ▷ Find block diagonal matrix in (1.1)
8: $Z \leftarrow \mathbf{0}_{n\times n}$
9: **if** $k_1 \neq 0$ **then**
10: $M^{-1} \leftarrow \text{inverse}(V_{:k_1,:k_1})$
11: $Z_{:k_1,:k_1} \leftarrow M^{-1}$
12: **return** UZU^{-1}

Problem 2. Write a function that accepts an $n \times n$ matrix A and a tolerance for rounding eigenvalues to zero. Use Algorithm 1.1 to compute the Drazin inverse A^D. Use your function from Problem 1 to verify your implementation.

ACHTUNG!

Because the algorithm for the Drazin inverse requires calculation of the inverse of a matrix, it is unstable when that matrix has a high condition number. If the algorithm does not find the correct Drazin inverse, check the condition number of V from Algorithm 1.1

NOTE

The Drazin inverse is called a *pseudoinverse* because $A^D = A^{-1}$ for invertible A, and for noninvertible A, A^D always exists and acts similarly to an inverse. There are other matrix pseudoinverses that preserve different qualities of A, including the *Moore-Penrose pseudoinverse* $A^\dagger$, which can be thought of as the least squares approximation to A^{-1}.

Applications of the Drazin Inverse

Effective Resistance

The *effective resistance* between two nodes in a undirected graph is a measure of how connected those nodes are. The concept originates from the study of circuits to measure the resistance between two points on the circuit. A *resistor* is a device in a circuit which limits or regulates the flow of electricity. Two points that have more resistors between them have more resistance, while those with fewer resistors between them have less resistance. The entire circuit can be represented by a graph where the nodes are the points of interest and the number of edges connecting two nodes indicates the number of resistors between the corresponding points. See Figure 1.1 for an example.

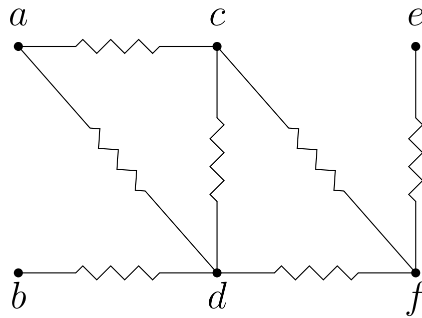

Figure 1.1: A graph with a resistor on each edge.

In electromagnetism, there are rules for manually calculating the effective resistance between two nodes for relatively simple graphs. However, this is infeasible for large or complicated graphs. Instead, we can use the Drazin inverse to calculate effective resistance for any graph.

First, create the *adjacency matrix*[1] of the graph, the matrix where the (ij)th entry is the number of connections from node i to node j. Next, calculate the Laplacian L of the adjacency matrix. Then if R_{ij} is the effective resistance from node i to node j,

$$R_{ij} = \begin{cases} (\widetilde{L}^j)^D_{ii} & \text{if } i \neq j \\ 0 & \text{if } i = j, \end{cases} \tag{1.4}$$

where $\widetilde{L}^j$ is the Laplacian with the jth row of the Laplacian replaced by the jth row of the identity matrix, and $(\widetilde{L}^j)^D$ is its Drazin inverse.

Problem 3. Write a function that accepts the $n \times n$ adjacency matrix of an undirected graph. Use (1.4) to compute the effective resistance from each node to every other node. Return an $n \times n$ matrix where the (ij)th entry is the effective resistance from node i to node j. Keep the following in mind:

- The resulting matrix should be symmetric.
- The effective resistance from a node to itself is 0.

[1]See Problem 1 of Image Segmentation for a refresher on adjacency matrices and the Laplacian.

- Consider creating the matrix column by column instead of entry by entry. Every time you compute the Drazin inverse, the whole diagonal of the matrix can be used.

Test your function using the graphs and values from Figure 1.2.

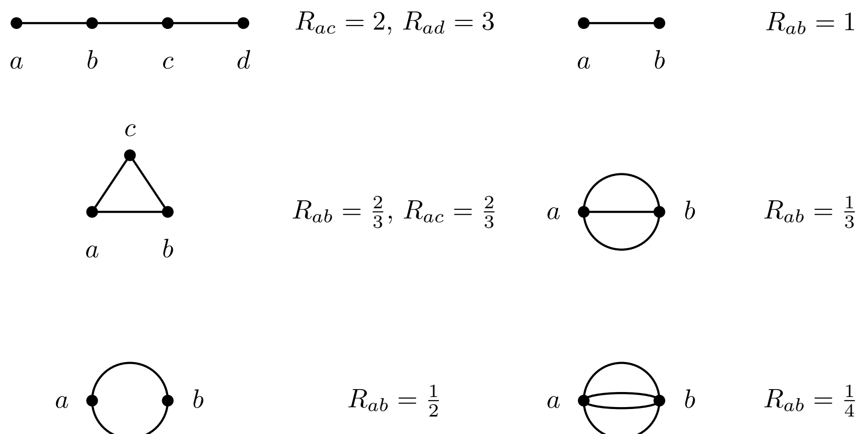

Figure 1.2: The effective resistance between two points for several simple graphs. Nodes that are farther apart have a larger effective resistance, while nodes that are nearer or better connected have a smaller effective resistance.

Link Prediction

Link prediction is the problem of predicting the likelihood of a future association between two unconnected nodes in a graph. Link prediction has application in many fields, but the canonical example is friend suggestions on Facebook. The Facebook network can be represented by a large graph where each user is a node, and two nodes have an edge connecting them if they are "friends." Facebook aims to predict who you would like to become friends with in the future, based on who you are friends with now, as well as discover which friends you may have in real life that you have not yet connected with online. To do this, Facebook must have some way to measure how closely two users are connected.

We will compute link prediction using effective resistance as a metric. Effective resistance measures how closely two nodes are connected, and nodes that are closely connected at present are more likely to be connected in the future. Given an undirected graph, the next link should connect the two unconnected nodes with the least effective resistance between them.

Problem 4. Write a class called `LinkPredictor` for performing link prediction. Implement the `__init__()` method so that it accepts the name of a `csv` file containing information about a social network. Each row of the file should contain the names of two nodes which are connected by an (undirected) edge.

Store each of the names of the nodes of the graph as an ordered list. Next, create the adjacency matrix for the network where the ith row and column of the matrix correspond to the

ith member of the list of node names. Finally, use your function from Problem 3 to compute the effective resistance matrix. Save the list of names, the adjacency matrix, and the effective resistance matrix as attributes.

Problem 5. Implement the following methods in the `LinkPredictor` class:

1. `predict_link()`: Accept a parameter `node` which is either `None` or a string representing a node in the network. If `node` is `None`, return a tuple with the names of the nodes between which the next link should occur. However, if `node` is a string, return the name of the node which should be connected to `node` next out of all other nodes in the network. If `node` is not in the network, raise a `ValueError`. Take the following into consideration:

   (a) You want to find the two nodes which have the smallest effective resistance between them which are not yet connected. Use information from the adjacency matrix to zero out all entries of the effective resistance matrix that represent connected nodes. The "`*`" operator multiplies arrays component-wise, which may be helpful.

   (b) Find the next link by finding the minimum value of the array that is nonzero. Your array may be the whole matrix or just a column if you are only considering links for a certain node. This can be accomplished by passing `np.min()` a masked version of your matrix to exclude entries that are 0.

   (c) NumPy's `np.where()` is useful for finding the minimum value in an array:

```
>>> A = np.random.randint(-9,9,(3,3))
>>> A
array([[ 6, -8, -9],
       [-2,  1, -1],
       [ 4,  0, -3]])

# Find the minimum value in the array.
>>> minval = np.min(A)
>>> minval
-9

# Find the location of the minimum value.
>>> loc = np.where(A==minval)
>>> loc
(array([0], dtype=int64), array([2], dtype=int64))
```

2. `add_link()`: Take as input two names of nodes, and add a link between them. If either name is not in the network, raise a `ValueError`. Add the link by updating the adjacency matrix and the effective resistance matrix.

Figure 1.3 visualizes the data in `social_network.csv`. Use this graph to verify that your class is suggesting plausible new links. You should observe the following:

- In the entire network, Emily and Oliver are most likely to become friends next.
- Melanie is predicted to become friends with Carol next.
- Alan is expected to become friends with Sonia, then with Piers, and then with Abigail.

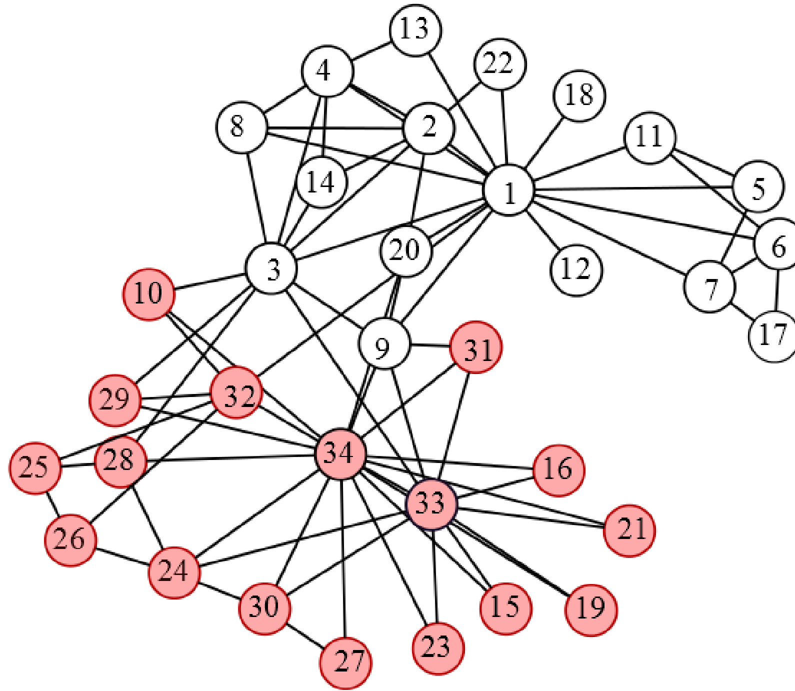

Figure 1.3: The social network contained in `social_network.csv`. Adapted from data by Wayne. W Zachary (see `https://en.wikipedia.org/wiki/Zachary%27s_karate_club`).

1. Piers
2. Abigail
3. Oliver
4. Stephanie
5. Carol
6. Melanie
7. Stephen
8. Sally
9. Penelope
10. Alan
11. Trevor
12. Jake
13. Mary
14. Anna
15. Ruth
16. Evan
17. Connor
18. John
19. Max
20. Eric
21. Theresa
22. Paul
23. Alexander
24. Colin
25. Jake
26. Jane
27. Brandon
28. Thomas
29. Christopher
30. Charles
31. Madeleine
32. Tracey
33. Sonia
34. Emily