Free_HW

June 14, 2023

0.0.1 Free Homework

Implement the Inverse Power Method and the Deflation Method to compute the M smallest eigenvalues of a symmetric matrix.

Importing the necessary libraries The only library used in this notebook is `Numpy`

[ ]:
```
import numpy as np
```

[ ]:
```
#ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

In order to test our code, we have created an irevertable matrix of size 5x5 We have also created the initial eigenvector and eigenvalue in order to use in the inverse power method.

[ ]:
```
size = 5  # Size of the matrix

# Generate a random matrix
matrix = np.random.rand(size, size)

# Make the matrix symmetric
matrix = (matrix + matrix.T) / 2

# Add diagonal dominance
diagonal = np.diag(np.sum(np.abs(matrix), axis=1))
matrix = matrix + diagonal

# Add positive definiteness
eigenvalues, _ = np.linalg.eig(matrix)
min_eigenvalue = np.min(eigenvalues)
matrix = matrix + np.eye(size) * (abs(min_eigenvalue) + 1)


# Generate a random initial eigenvector
initial_eigenvector = np.random.rand(size)

# Normalize the initial eigenvector
initial_eigenvector = initial_eigenvector / np.linalg.norm(initial_eigenvector)
```

```
#print(initial_eigenvector)
```

0.0.2 Inverse power method

In the following cell, we have implemented the inverse power method. The method takes as input the matrix, the initial eigenvector, the initial eigenvalue, the number of iterations and the tolerance. The method returns the final eigenvalue and eigenvector. The method works as follows: It begins by taking the size of the matrix. The eigenvalue and eigenvector variables are initialized to default values of 0.0 and None, respectively. The algorithm enters a loop that runs for a maximum of max_iterations times. Inside the loop, it attempts to solve the linear system using matrix inversion. It calculates the inverse of the matrix by subtracting the target eigenvalue multiplied by the identity matrix and then using `np.linalg.inv` to compute the inverse. If the matrix is singular and cannot be inverted, a `LinAlgError` is caught, and the method returns `None` value for both the eigenvalue and eigenvector. If the linear system is solvable, the algorithm computes the next eigenvector approximation by multiplying the inverted matrix with the initial eigenvector using np.dot. The next eigenvector is then normalized to ensure it has unit length using `np.linalg.norm`. To approximate the eigenvalue, the algorithm computes the dot product of the next eigenvector with the matrix product of the matrix and the transposed next eigenvector using `np.dot`. The algorithm checks for convergence by comparing the absolute difference between the new eigenvalue approximation and the previous eigenvalue with the specified tolerance. If the difference is below the tolerance, the algorithm considers the iteration converged and breaks out of the loop. If convergence is not achieved, the eigenvalue and eigenvector are updated with the new values, and the algorithm proceeds to the next iteration. The initial eigenvector is updated with the next eigenvector for the next iteration. Finally, when the algorithm terminates (either due to convergence or reaching the maximum iterations), it returns the converged eigenvalue and eigenvector.

```
[ ]: def inverse_power_method(matrix, initial_eigenvector ,target_value,␣
      ↪max_iterations=100, tolerance=1e-6):
         n = matrix.shape[0]


         eigenvalue = 0.0  # to initialize eigenvalue with a default value
         eigenvector = None

         for iteration in range(max_iterations):
             # Solving the linear system using LU decomposition
             try:
                 inverted_matrix = np.linalg.inv(matrix - target_value * np.eye(n))
                 next_vector = np.dot(inverted_matrix, initial_eigenvector)
             except np.linalg.LinAlgError:
                 print("Matrix is singular. Inverse power method failed.")
                 return None, None

             # Normalizing the next vector
             next_vector /= np.linalg.norm(next_vector)
```

```
        # Compute the eigenvalue approximation
        eigenvalue_next = np.dot(np.dot(next_vector, matrix), next_vector.
 ↪transpose())

        # Checking for convergence
        if np.abs(eigenvalue_next - eigenvalue) < tolerance:
            eigenvalue = eigenvalue_next
            eigenvector = next_vector
            break

        # Updating the eigenvalue and eigenvector
        eigenvalue = eigenvalue_next
        eigenvector = next_vector

        # Update the initial vector for the next iteration
        initial_eigenvector = next_vector

    return eigenvalue, eigenvector
```

[ ]:
```
#using the inverse power method to find the smallest eigenvalue
eigenvalue, eigenvector = inverse_power_method(matrix, initial_eigenvector,␣
 ↪1e-3)
print("Eigenvalue:", eigenvalue)
```

```
Eigenvalue: 4.25998252193301
```

[ ]:
```
# to calculate the eigenvalues and eigenvectors of matrix using NumPy eigsh␣
 ↪function in order to test our implementation of the inverse power method
eigenvalues, eigenvectors = np.linalg.eigh(matrix)
print("Eigenvalues:", eigenvalues)
```

```
Eigenvalues: [4.25997883 4.72091611 5.51419501 5.53371377 7.60313517]
```

0.0.3 Deflation method

We first use the power method in order to find the dominant eigenvalue and eigenvector. Then use the output of the power method in the deflation method in order to find the smallest eigenvalue and eigenvector.

In this part we first implement the power method. The power_method function implements the power method algorithm to estimate the dominant eigenvalue of a given matrix. The function takes as input the matrix, the initial eigenvector, the number of iterations, and the tolerance. The algorithm works as follows: The function begins by extracting the size of the matrix (assuming it's square) using `matrix.shape[0]`. It initializes the eigenvector randomly using np.random.rand to generate random values between 0 and 1. The eigenvector is then normalized to have unit length using `np.linalg.norm`. The algorithm enters a loop that runs for a maximum of max_iterations times. Inside the loop, it computes a new eigenvector approximation by multiplying the matrix with the current eigenvector using `np.dot`. This step essentially represents the power method's iterative process. The new eigenvector is then normalized to have unit length using `np.linalg.norm`. The algorithm checks for convergence by comparing the norm (Euclidean distance) between the current eigenvector and the new eigenvector with the specified tolerance. If the difference falls below the tolerance, the algorithm considers the iteration converged and breaks out of the loop. If convergence is not achieved, the new eigenvector becomes the current eigenvector, and the algorithm proceeds to the next iteration. After the loop terminates, the estimated dominant eigenvalue is computed using the formula:

$$\text{eigenvalue} = (new\_eigenvector^{\top}) \cdot matrix \cdot new\_eigenvector$$

where ^{T} denotes the transpose operation and $\cdot$ represents matrix multiplication.

[ ]:
```python
# Power method
def power_method(matrix, tolerance, max_iterations):
    size = matrix.shape[0]

    # Random initial eigenvector
    eigenvector = np.random.rand(size)
    eigenvector = eigenvector / np.linalg.norm(eigenvector)

    for _ in range(max_iterations):
        new_eigenvector = np.dot(matrix, eigenvector)
        new_eigenvector = new_eigenvector / np.linalg.norm(new_eigenvector)

        # Check convergence
        if np.linalg.norm(eigenvector - new_eigenvector) < tolerance:
            break

        eigenvector = new_eigenvector

    eigenvalue = np.dot(np.dot(new_eigenvector.T, matrix), new_eigenvector)

    return eigenvalue
```

Now, in the following cell we implement the deflation method in which we use the output of the power method. The deflation function implements the deflation method for reducing the dimensionality of a matrix after computing an eigenvalue and eigenvector pair. The algorithm works as follows: The function begins by extracting the size of the matrix (assuming it's square) using `matrix.shape[0]`. The eigenvector is converted to a `NumPy` array using `np.array`. The outer product of the eigenvector is computed using `np.outer`. This creates a matrix, denoted as `P`, where each element `P[i][j]` is the product of the `i-th` element of the eigenvector and the `j-th` element of the eigenvector. The matrix is updated with deflation by subtracting the product of the eigenvalue and `P` from the original matrix. This step modifies the matrix to remove the influence of the computed eigenvalue and eigenvector pair. The row and column corresponding to the computed eigenvector are removed from the matrix using `np.delete`. This is done to reduce the dimensionality of the matrix after deflation. In the provided code, the row and column with index 0 (assuming zero-based indexing) are removed. The modified matrix, after deflation, is returned as the output of the function.

```
[ ]: #deflation method
     def deflation(matrix, eigenvalue, eigenvector):
         size = matrix.shape[0]

         # Convert eigenvector to numpy array
         eigenvector = np.array(eigenvector)

         # Compute outer product
         P = np.outer(eigenvector, eigenvector)

         # Update matrix with deflation
         matrix -= eigenvalue * P

         # Remove row and column corresponding to the computed eigenvector
         matrix = np.delete(matrix, (0), axis=0)
         matrix = np.delete(matrix, (0), axis=1)

         return matrix
```

In the following cell, we create a function to use both the power method and the delation method in order to find the smallest eigenvalue. The function takes as input the matrix, the initial eigenvector, the number of iterations, and the tolerance. The function initializes an empty list called `eigenvalues` to store the eigenvalues obtained during the iteration. The function enters a loop that continues as long as the dimension of the matrix is greater than 1 (indicating there are more eigenvalues to compute). Inside the loop, the power method is used to compute an eigenvalue estimate by calling the `power_method` function. The computed eigenvalue is stored in the `eigenvalues` list. A random eigenvector is generated for the current dimension of the matrix using `np.random.rand`. The eigenvector is then normalized to have unit length using `np.linalg.norm`. The deflation method is applied to the matrix by calling the deflation function. The eigenvalue and eigenvector computed in the previous step are used to modify the matrix, removing the influence of the computed eigenvalue-eigenvector pair. The dimension of the matrix is reduced after the `deflation`, and the loop repeats for the next iteration if the new dimension is still greater than 1. After the loop terminates, the smallest eigenvalue is obtained by finding the minimum value in the `eigenvalues` list using `np.min`. The smallest eigenvalue is returned as the output of the function.

```
[ ]: #function to use both power method and deflation method to find the smallest␣
      ↪eigenvalue

     def find_smallest_eigenvalue(matrix, tolerance, max_iterations):
         eigenvalues = []

         while matrix.shape[0] > 1:
             eigenvalue = power_method(matrix, tolerance, max_iterations)

             eigenvalues.append(eigenvalue)

             eigenvector = np.random.rand(matrix.shape[0])
```

```
        eigenvector = eigenvector / np.linalg.norm(eigenvector)

        matrix = deflation(matrix, eigenvalue, eigenvector)

    smallest_eigenvalue = np.min(eigenvalues)
    return smallest_eigenvalue
```

In the following cell, we call the `find_smallest_eigenvalue` function in order to find the smallest eigenvalue of the matrix.

```
[ ]: #using the function to find the smallest eigenvalue
     smallest_eigenvalue = find_smallest_eigenvalue(matrix, tolerance=1e-3,␣
      ↪max_iterations=100)
     print("Smallest Eigenvalue:", smallest_eigenvalue)
```

```
Smallest Eigenvalue: 5.114666598539619
```