Brunton and Kutz Problem 7.8: Markov Chain Modeling and DMD

Source Filename: /main.py

Rico A. R. Picone

First, import the necessary libraries:

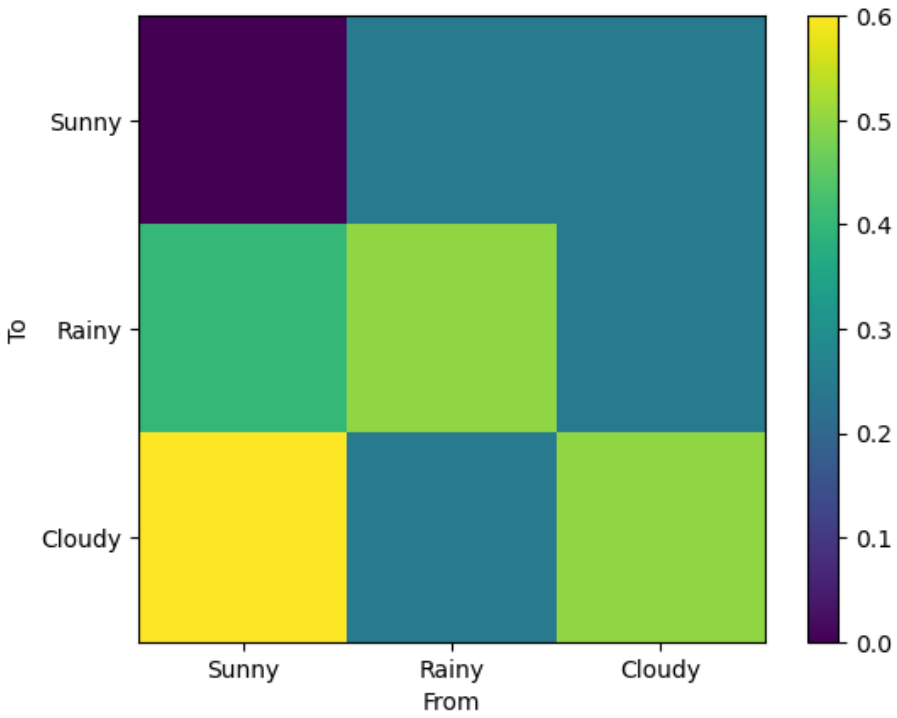
```
import numpy as np
import matplotlib.pyplot as plt
```

Next, define the Markov chain transition matrix:

```
P = np.array([
    [0.0, 0.25, 0.25], [0.40, 0.50, 0.25], [0.60, 0.25, 0.50]
])
```

As the footnote in the exercise states, this is the transpose of the more common Russian standard notation. This means that the (i, j) element of the matrix is the probability of transitioning from state j to state i. We can visualize the matrix using a heatmap:

```
fig, ax = plt.subplots()
im = ax.imshow(P, cmap="viridis")
ax.set_xticks([0, 1, 2])
ax.set_yticks([0, 1, 2])
ax.set_xticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_xlabel("From")
ax.set_ylabel("To")
fig.colorbar(im)
plt.draw()
```



We see that the probability of transitioning from a sunny day to a sunny day is 0, to a rainy day is 0.4, and to a cloudy day is 0.6. Note that the columns of the matrix sum to 1 (i.e., it has to be sunny, rainy, or cloudy at any given time step).

Long-term distribution

The long-term distribution of the Markov chain is the eigenvector of the transition matrix corresponding to the eigenvalue of 1. We can find this eigenvector using the `np.linalg.eig` function:

```
eigenvalues, eigenvectors = np.linalg.eig(P)
idx = np.argmin(np.abs(eigenvalues - 1.0))  # Index of e-val close to 1
v = np.real(eigenvectors[:, idx])  # Corresponding eigenvector
v = v / np.sum(v)  # Normalize eigenvector
print(f"Long-term distribution: {v}")
print(f"Interpretation:\nSunny probability:{v[0]:.2f}\n"+
      f"Rainy probability: {v[1]:.2f}\n"
      f"Cloudy probability: {v[2]:.2f}")
```

```
Long-term distribution: [0.2        0.37333333 0.42666667]
Interpretation:
Sunny probability:0.20
```

```
Rainy probability: 0.37
Cloudy probability: 0.43
```

Simulation

Define an observer function that samples the state of the Markov chain:

```python
def observer(x):
    """Observe the state of the Markov chain"""
    random_index = np.random.choice(range(3), p=x)
    x = np.zeros(x.shape)
    x[random_index] = 1.0
    return x
```

Simulate a random instance (i.e., one corresponding to a random initial condition) of the process, observing the state at each time step:

```python
T = 5000  # Number of time steps
x = np.zeros((T, 3))  # State at each time step
np.random.seed(0)  # Seed random number generator for reproducibility
initial_nonzero_index = np.random.randint(0, x.shape[1])
x[0, initial_nonzero_index] = 1.0  # Initial condition
for t in range(0, T-1):
    x_pre_observation = P @ x[t]
    x[t+1] = observer(x_pre_observation)
print(f"First 10 states: {x[:10]}")
```

```
First 10 states: [[1. 0. 0.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]
 [1. 0. 0.]
 [0. 1. 0.]]
```

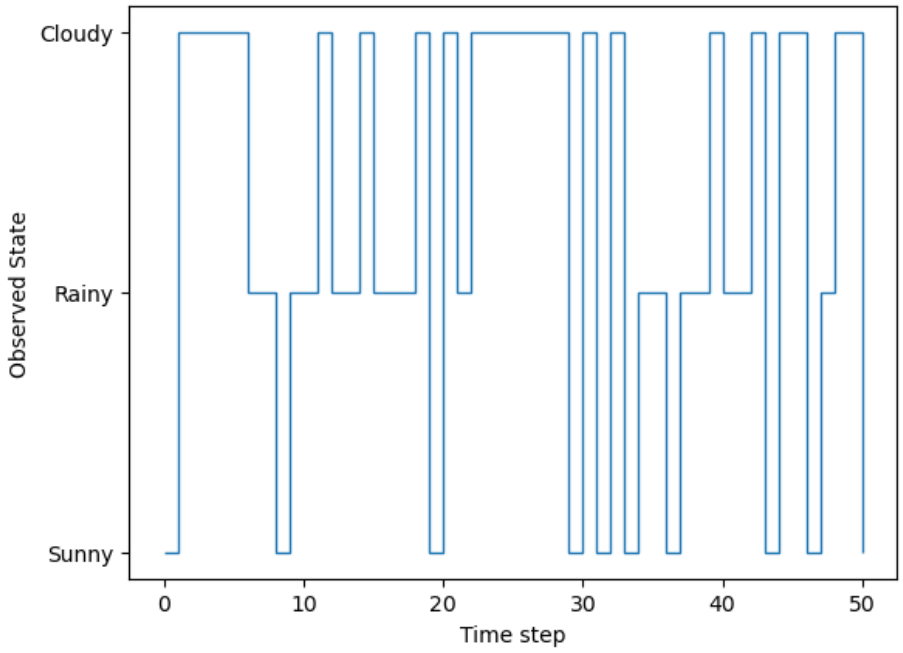
Visualize the simulation. First, convert the state to values that can be plotted:

```python
x_visualize = np.argmax(x, axis=1)  # Integer representation of states
```

Plot the first 50 observed states:

```python
n_plot = 50
fig, ax = plt.subplots()
ax.stairs(
    x_visualize[:n_plot],
    edges=np.arange(0, len(x_visualize[:n_plot])+1)
)
```

```
ax.set_xlabel("Time step")
ax.set_ylabel("Observed State")
ax.set_yticks([0, 1, 2])
ax.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
ax.set_ylim(-0.1, 2.1)
plt.draw()
```



Dynamic Mode Decomposition (DMD)

Define the exact DMD function from Brunton and Kutz (2022):

```
def DMD(X,Xprime,r):
    # Step 1
    U, Sigma, VT = np.linalg.svd(X, full_matrices=0)
    Ur = U[:, :r]
    Sigmar = np.diag(Sigma[:r])
    VTr = VT[:r, :]
    # Step 2
    Atilde = np.linalg.solve(Sigmar.T, (Ur.T @ Xprime @ VTr.T).T).T
    # Step 3
    Lambda, W = np.linalg.eig(Atilde)
    Lambda = np.diag(Lambda)
    # Step 4
    Phi = Xprime @ np.linalg.solve(Sigmar.T, VTr).T @ W
```

```
    alpha1 = Sigmar @ VTr[:,0]
    b = np.linalg.solve(W @ Lambda, alpha1)
    return Phi, Lambda, b
```

Construct the data matrices X and X':

```
X = x[:-1].T
Xprime = x[1:].T
```

Compute the DMD modes:

```
r = 3  # Number of modes
Phi, Lambda, b = DMD(X, Xprime, r)
print(f"Phi:\n{Phi}")
print(f"Lambda:\n{Lambda}")
print(f"b:\n{b}")
```

```
Phi:
[[-0.33268176 -0.01006597 -0.20196528]
 [-0.61594571  0.18226477  0.05599596]
 [-0.71409645 -0.1721988   0.14596931]]
Lambda:
[[ 1.          0.          0.        ]
 [ 0.          0.25094661  0.        ]
 [ 0.          0.         -0.25540666]]
b:
[-0.60142276 -0.82832248 -3.91938537]
```

Reconstruct the P matrix from the DMD modes:

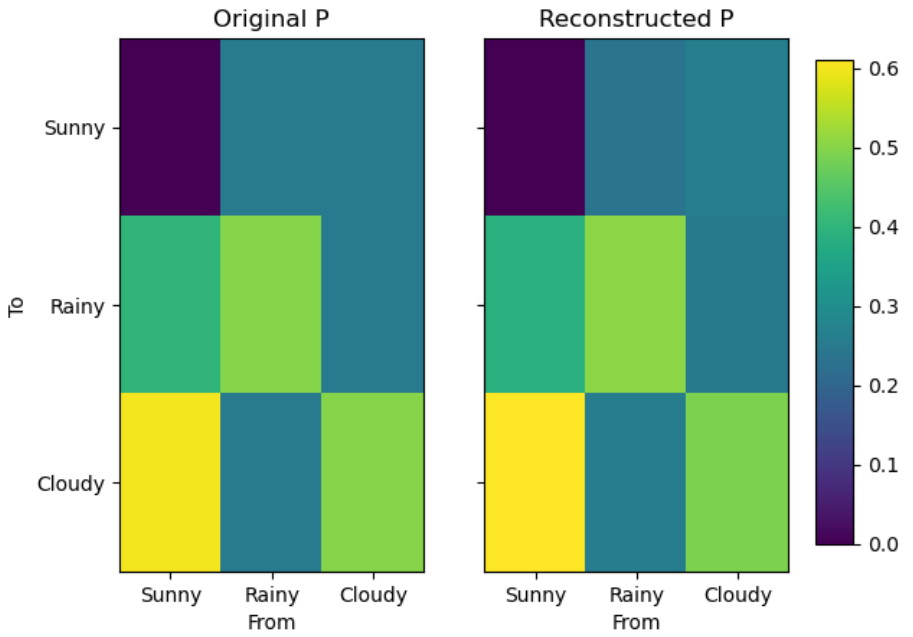
```
print(f"Phi shape: {Phi.shape}")
print(f"Lambda shape: {Lambda.shape}")
P_dmd = Phi @ Lambda @ np.linalg.inv(Phi)
print(f"P:\n{P}")
print("P_dmd:\n" +
    np.array2string(
        P_dmd, precision=2,
        floatmode="fixed", suppress_small=True
    )
)
```

```
Phi shape: (3, 3)
Lambda shape: (3, 3)
P:
[[0.   0.25 0.25]
 [0.4  0.5  0.25]
 [0.6  0.25 0.5 ]]
P_dmd:
[[0.00 0.23 0.26]
 [0.39 0.50 0.25]
```

```
 [0.61 0.26 0.49]]
```

Visually compare the original and reconstructed transition matrices:

```
zmin = min(np.min(P), np.min(P_dmd))
zmax = max(np.max(P), np.max(P_dmd))
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True)
im0 = ax[0].imshow(
    P, cmap="viridis", vmin=zmin, vmax=zmax, aspect='auto'
)
ax[0].set_title("Original P")
im1 = ax[1].imshow(
    P_dmd, cmap="viridis", vmin=zmin, vmax=zmax, aspect='auto'
)
ax[1].set_title("Reconstructed P")
ax[0].set_ylabel("To")
for a in ax:
    a.set_xticks([0, 1, 2])
    a.set_yticks([0, 1, 2])
    a.set_xticklabels(["Sunny", "Rainy", "Cloudy"])
    a.set_yticklabels(["Sunny", "Rainy", "Cloudy"])
    a.set_xlabel("From")
    a.set_xlim(-0.5, 2.5)
    a.set_ylim(2.5, -0.5)
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.88, 0.15, 0.04, 0.7])
fig.colorbar(im1, cax=cbar_ax)
plt.show()
```



We appear to achieve a good reconstruction of the transition matrix using DMD. The DMD modes can be used to predict the future state of the system.

Perhaps a numerical evaluation of the prediction error would be useful. One approach is to compute the Eucldean distance between the eigenvectors of the original and reconstructed transition matrices. We have already computed both sets of eigenvectors, so we can proceed to write a function to compare the eigenvectors of two matrices and compute the error. The function will normalize the eigenvectors and compute the Euclidean distance between them. However, the ordering of the eigenvectors may differ between the two matrices, so we will need to account for this:

```
def eigenvector_error(M1: np.ndarray, M2: np.ndarray):
    """Compute the distance between eigenvectors of two matrices

    The eigenvectors are normalized before computing the error.
    The error corresponds to the Euclidean distance between the
    eigenvectors of two matrices. The indexing and sign of the
    eigenvectors may differ between the two matrices, so the
    columns are matched. The output is the error for each,
    using the index of the M1 eigenvectors.

    Parameters
    ----------
    M1 : np.ndarray
```

```
    Matrix of eigenvectors
M2 : np.ndarray
    Matrix of eigenvectors

Returns
-------
error : np.ndarray
    Error for each eigenvector
M1_out : np.ndarray
    Normalized matrix of eigenvectors from M1
M2_out : np.ndarray
    Normalized and potentially reordered (and sign-flipped)
    matrix of eigenvectors from M2
"""
M1 = M1 / np.linalg.norm(M1, ord=2, axis=0)  # Normalize e-vecs
M2 = M2 / np.linalg.norm(M2, ord=2, axis=0)
M2_out = M2.copy()  # Copy M2 for matching
# Match e-vecs of M1 to e-vecs of M2
error = np.zeros(M1.shape[1])  # Initialize error
used_indices = []  # Indices of M2 columns already matched
for i in range(M1.shape[1]):  # Loop over columns of M1
    v = M1[:, [i]]  # Current e-vec of M1
    distances1 = np.linalg.norm(M2 - v, ord=2, axis=0)
    distances2 = np.linalg.norm(M2 + v, ord=2, axis=0)
    distances = np.vstack([distances1, distances2])
    di_min = np.unravel_index(
        np.argmin(distances, axis=None), distances.shape
    )  # Index of minimum distance
    if di_min[1] in used_indices:  # Column already matched
        raise RuntimeError(
            f"Column {di_min[1]} already matched so 2 columns" +
            "are too close for this method"
        )
    else:
        used_indices.append(di_min[1])
    if di_min[1] != i:  # Columns do not match
        # Swap columns of M2 to match e-vecs of M1
        M2_out[:, [i, di_min[1]]] = \
            M2[:, [di_min[1], i]]
    if di_min[0] == 0:  # No sign change
        error[i] = distances1[di_min[1]]
    else:  # Sign change
        M2_out[:, i] = -M2_out[:, i]
        error[i] = distances2[di_min[1]]
return error, M1, M2_out
```

Now apply the function to the eigenvectors of the original transition matrix and the DMD eigenvectors of Φ:

```
error, MP, MP_dmd = eigenvector_error(eigenvectors, Phi)
print("Eigenvector errors (percent): " +
    f"{np.array2string(error*100, precision=2, suppress_small=True)}"
)
print("Original eigenvectors (noramlized):\n" +
    f"{np.array2string(MP, precision=2, suppress_small=True)}"
)
print("Original eigenvectors (noramlized, reconstructed):\n" +
    f"{np.array2string(MP_dmd, precision=2, suppress_small=True)}"
)
```

```
Eigenvector errors (percent): [0.67 2.48 4.91]
Original eigenvectors (noramlized):
[[-0.33 -0.8   0.  ]
 [-0.62  0.24 -0.71]
 [-0.71  0.56  0.71]]
Original eigenvectors (noramlized, reconstructed):
[[-0.33 -0.79  0.04]
 [-0.62  0.22 -0.73]
 [-0.71  0.57  0.69]]
```

Check to see if the columns of the reconstructed transition matrix sum to 1, as they should:

```
print("Sum of columns of P_dmd: " +
    np.array2string(np.sum(P_dmd, axis=0), precision=16)
)
```

```
Sum of columns of P_dmd: [1.0000000000000022 0.9999999999999997 0.9999999999999599]
```

These sums are remarkably close to 1, which means that the reconstructed transition matrix is a valid Markov chain transition matrix.