Assignment4_9628_jzeiders

November 11, 2024

1 John Zeiders (jzeiders) - Assignment 4

[ ]:
```python
import numpy as np
from numpy.linalg import inv, det
import pandas as pd
```

2 Part 1

[ ]:
```python
def Estep(data, prob, mean, Sigma):
    """
    E-step: Calculate responsibilities

    Parameters:
    -----------
    data: ndarray, shape (n, p)
        The input data matrix
    prob: ndarray, shape (G,)
        Mixing proportions
    mean: ndarray, shape (p, G)
        Mean vectors for each component
    Sigma: ndarray, shape (p, p)
        Shared covariance matrix

    Returns:
    --------
    responsibilities: ndarray, shape (n, G)
        Matrix of posterior probabilities P(Z_i=k|x_i)
    """
    n, p  = data.shape
    G = prob.shape[0]

    # Initialize matrix to store densities
    densities = np.zeros((n, G))

    # Precompute constants related to the covariance matrix
    try:
        inv_Sigma = np.linalg.inv(Sigma)  # Inverse of covariance matrix
        det_Sigma = np.linalg.det(Sigma)  # Determinant of covariance matrix
        if det_Sigma <= 0:
            raise np.linalg.LinAlgError("Covariance matrix is not positive␣
↪definite.")
    except np.linalg.LinAlgError as e:
        raise ValueError("Invalid covariance matrix. " + str(e))

    # Compute the normalization constant for the multivariate normal␣
↪distribution
    norm_const = 1.0 / (np.power((2 * np.pi), p / 2) * np.sqrt(det_Sigma))

    # Compute densities for each Gaussian component
    for k in range(G):
        # Extract the mean vector for the k-th component
        mu_k = mean[:, k]  # Shape: (p,)

        # Compute the difference between each data point and the mean vector
        diff = data - mu_k  # Shape: (n, p)

        # Compute the exponent term for the multivariate normal PDF
        # This represents: -0.5 * (x - mu_k)^T * inv_Sigma * (x - mu_k)
        # Efficiently computed using einsum
        exponent = -0.5 * np.einsum('ij,jk,ik->i', diff, inv_Sigma, diff)  #␣
↪Shape: (n,)

        # Compute the density for the k-th component
        densities[:, k] = norm_const * np.exp(exponent)  # Shape: (n,)

    # Multiply by mixing proportions
    weighted_densities = densities * prob[np.newaxis, :]

    # Normalize to get responsibilities
    responsibilities = weighted_densities / np.sum(weighted_densities, axis=1)[:
↪, np.newaxis]

    return responsibilities

def Mstep(data, responsibilities):
    """
    M-step: Update parameters

    Parameters:
    -----------
    data: ndarray, shape (n, p)
        The input data matrix
    responsibilities: ndarray, shape (n, G)
        Matrix of posterior probabilities
    Returns:
    --------
    prob: ndarray, shape (G,)
        Updated mixing proportions
    mean: ndarray, shape (p, G)
        Updated mean vectors
    Sigma: ndarray, shape (p, p)
        Updated shared covariance matrix
    """
    n, p = data.shape
    G = responsibilities.shape[1]

    # Update mixing proportions
    prob = np.mean(responsibilities, axis=0)

    # Update means
    mean = np.zeros((p, G))
    for k in range(G):
        mean[:, k] = np.sum(responsibilities[:, k:k+1] * data, axis=0) / np.
 ↪sum(responsibilities[:, k])

    # Update shared covariance matrix
    Sigma = np.zeros((p, p))
    for k in range(G):
        diff = data - mean[:, k]  # Shape: (n, p)
        weighted_diff = responsibilities[:, k].reshape(n, 1) * diff  # Shape:␣
 ↪(n, p)
        Sigma += weighted_diff.T @ diff  # Shape: (p, p)
    Sigma /= n

    return prob, mean, Sigma

def loglik(data, prob, mean, Sigma):
    """
    Compute log-likelihood

    Parameters:
    -----------
    data: ndarray, shape (n, p)
        The input data matrix
    prob: ndarray, shape (G,)
        Mixing proportions
    mean: ndarray, shape (p, G)
        Mean vectors
    Sigma: ndarray, shape (p, p)
        Shared covariance matrix
    Returns:
    --------
    ll: float
        Log-likelihood value
    """
    n = data.shape[0]
    G = prob.shape[0]

    # Initialize array for component densities
    densities = np.zeros((n, G))
    d = data.shape[1]
    inv_Sigma = np.linalg.inv(Sigma)
    det_Sigma = np.linalg.det(Sigma)
    norm_const = 1.0 / (np.power((2 * np.pi), d / 2) * np.sqrt(det_Sigma))

    # Compute densities for each component
    for k in range(G):
            # Compute the difference between data and the k-th mean
        diff = data - mean[:, k]  # Shape: (n_samples, d)

        # Compute the exponent term: -0.5 * (diff @ inv_Sigma * diff).
 ↪sum(axis=1)
        # Efficient computation using einsum for element-wise multiplication␣
 ↪and summation
        exponent = -0.5 * (diff @ inv_Sigma * diff).sum(axis=1)

        # Compute the density for the k-th component
        densities[:, k] = prob[k] * norm_const * np.exp(exponent)

    # Sum over components and take log
    ll = np.sum(np.log(np.sum(densities, axis=1)))

    return ll

def myEM(data, G, initial_params, itmax):
    """
    Main EM algorithm function

    Parameters:
    -----------
    data: ndarray, shape (n, p)
        The input data matrix
    G: int
        Number of components
    initial_params: dict
        Dictionary containing initial parameters:
```

```
        'prob': mixing proportions
        'mean': mean vectors
        'Sigma': covariance matrix
    itmax: int
        Maximum number of iterations

    Returns:
    --------
    prob: ndarray, shape (G,)
        Final mixing proportions
    mean: ndarray, shape (p, G)
        Final mean vectors
    Sigma: ndarray, shape (p, p)
        Final shared covariance matrix
    ll: float
        Final log-likelihood value
    """
    # Initialize parameters
    prob = initial_params['prob']
    mean = initial_params['mean']
    Sigma = initial_params['Sigma']

    for _ in range(itmax):
        # E-step
        responsibilities = Estep(data, prob, mean, Sigma)

        # M-step
        prob, mean, Sigma = Mstep(data, responsibilities)

    # Compute final log-likelihood
    ll = loglik(data, prob, mean, Sigma)

    return prob, mean, Sigma, ll
```

[ ]:
```
def test_G2(data):
    """Test EM algorithm with G=2 components"""
    n, p = data.shape

    # Set initial parameters as specified
    p1 = 10/n
    prob = np.array([p1, 1-p1])

    mean = np.zeros((p, 2))
    mean[:, 0] = np.mean(data[:10], axis=0)  # mean of first 10 samples
    mean[:, 1] = np.mean(data[10:], axis=0)  # mean of remaining samples

    # Calculate initial Sigma
    diff1 = data[:10] - mean[:, 0]
    diff2 = data[10:] - mean[:, 1]
    Sigma = (diff1.T @ diff1 + diff2.T @ diff2) / n

    initial_params = {
        'prob': prob,
        'mean': mean,
        'Sigma': Sigma
    }

    # Run EM algorithm
    final_prob, final_mean, final_Sigma, final_ll = myEM(data, 2,␣
↪initial_params, 20)

    return final_prob, final_mean, final_Sigma, final_ll

def test_G3(data):
    """Test EM algorithm with G=3 components"""
    n, p = data.shape

    # Set initial parameters
    p1, p2 = 10/n, 20/n
    prob = np.array([p1, p2, 1-p1-p2])

    mean = np.zeros((p, 3))
    mean[:, 0] = np.mean(data[:10], axis=0)      # mean of first 10 samples
    mean[:, 1] = np.mean(data[10:30], axis=0)    # mean of next 20 samples
    mean[:, 2] = np.mean(data[30:], axis=0)      # mean of remaining samples

    # Calculate initial Sigma
    diff1 = data[:10] - mean[:, 0]
    diff2 = data[10:30] - mean[:, 1]
    diff3 = data[30:] - mean[:, 2]
    Sigma = (diff1.T @ diff1 + diff2.T @ diff2 + diff3.T @ diff3) / n

    initial_params = {
        'prob': prob,
        'mean': mean,
        'Sigma': Sigma
    }

    # Run EM algorithm
    final_prob, final_mean, final_Sigma, final_ll = myEM(data, 3,␣
↪initial_params, 20)

    return final_prob, final_mean, final_Sigma, final_ll
```

```
[ ]: # Load faithful dataset
     file_path = 'https://liangfgithub.github.io/Data/faithful.dat'  # Update this␣
      ↪path as necessary
     data = np.loadtxt(file_path, skiprows=1, dtype=float, usecols=(1,2))
     print("Data Shape:", data.shape)
     print("First 5 Data Points:\n", data[:5])
```

```
Data Shape: (272, 2)
First 5 Data Points:
 [[ 3.6    79.   ]
 [ 1.8    54.   ]
 [ 3.333  74.   ]
 [ 2.283  62.   ]
 [ 4.533  85.   ]]
```

```
[ ]: # Test G=2
     print("\nTesting with G=2:")
     prob2, mean2, Sigma2, ll2 = test_G2(data)

     print("\nprob")
     print(prob2)
     print("\nmean")
     print(mean2)
     print("\nSigma")
     print(Sigma2)
     print("\nloglik")
     print(ll2)
```

```

Testing with G=2:

prob
[0.04297883 0.95702117]

mean
[[ 3.49564188  3.48743016]
 [76.79789154 70.63205853]]

Sigma
[[  1.29793612  13.92433626]
 [ 13.92433626 182.58009247]]

loglik
-1289.5693549424104
```

```
[ ]: # Test G=3
     print("\nTesting with G=3:")
     prob3, mean3, Sigma3, ll3 = test_G3(data)
```

```python
print("\nprob")
print(prob3)
print("\nmean")
print(mean3)
print("\nSigma")
print(Sigma3)
print("\nloglik")
print(ll3)
```

```
Testing with G=3:

prob
[0.04363422 0.07718656 0.87917922]

mean
[[ 3.51006918  2.81616674  3.54564083]
 [77.10563811 63.35752634 71.25084801]]

Sigma
[[  1.26015772  13.51153756]
 [ 13.51153756 177.96419105]]

loglik
-1289.350958862739
```

3 Part 2

[ ]:
```python
def forward_pass(data, w, A, B):
    """
    Compute forward probabilities alpha(t,j)

    Args:
        data: T-by-1 observation sequence (1D array)
        w: Initial state distribution (mz,)
        A: Transition matrix (mz-by-mz)
        B: Emission matrix (mz-by-mx)

    Returns:
        alpha: Forward probabilities (T-by-mz)
    """
    T = len(data)
    mz = len(w)
    alpha = np.zeros((T, mz))

    # Initialize first time step
    alpha[0, :] = w * B[:, data[0]]

    # Forward recursion
    for t in range(1, T):
        alpha[t, :] = np.dot(alpha[t - 1, :], A) * B[:, data[t]].T

    return alpha

def backward_pass(data, A, B):
    """
    Compute backward probabilities beta(t,j)

    Args:
        data: T-by-1 observation sequence (1D array)
        A: Transition matrix (mz-by-mz)
        B: Emission matrix (mz-by-mx)

    Returns:
        beta: Backward probabilities (T-by-mz)
    """
    T = len(data)
    mz = A.shape[0]
    beta = np.zeros((T, mz))

    # Initialize last time step
    beta[-1, :] = 1

    # Backward recursion
    for t in range(T - 2, -1, -1):
        beta[t, :] = np.dot(A, (B[:, data[t + 1]] * beta[t + 1, :]))

    return beta

def BW_onestep(data, w, A, B):
    """
    One step of the Baum-Welch algorithm (E-step + M-step)

    Args:
        data: T-by-1 observation sequence (1D array)
        w: Initial state distribution (mz,)
        A: Current transition matrix (mz-by-mz)
        B: Current emission matrix (mz-by-mx)

    Returns:
        A_new: Updated transition matrix
        B_new: Updated emission matrix
    """
    T = len(data)
    mz = A.shape[0]
    mx = B.shape[1]

    # E-step: Compute forward and backward probabilities
    alpha = forward_pass(data, w, A, B)
    beta = backward_pass(data, A, B)

    # Compute xi(t,i,j) = P(z_t=i, z_{t+1}=j | x_{1:T})
    xi = np.zeros((T - 1, mz, mz))
    for t in range(T - 1):
        denominator = np.dot(np.dot(alpha[t, :], A) * B[:, data[t + 1]], beta[t␣
↪+ 1, :])
        numerator = alpha[t, :, np.newaxis] * A * B[:, data[t + 1]] * beta[t +␣
↪1, :]
        xi[t, :, :] = numerator / denominator

    # Compute state probabilities (gamma)
    gammas_j = np.zeros((T, mz))
    gammas_j[:-1] = np.sum(xi, axis=2)
    # Fix for last time step - should use alpha and beta
    gammas_j[-1] = (alpha[-1] * beta[-1]) / np.sum(alpha[-1] * beta[-1])

    # M-step: Update parameters
    # Update A
    A_new = np.zeros_like(A)
    for i in range(mz):
        for j in range(mz):
            numerator = np.sum(xi[:, i, j])
            denominator = np.sum(xi[:, i, :])
            A_new[i, j] = numerator / denominator

    # Update B (vectorized version)
    B_new = np.zeros_like(B)
    for l in range(mx):
        mask = (data == l)
        B_new[:, l] = np.sum(gammas_j[mask], axis=0) / (np.sum(gammas_j,␣
↪axis=0) + 1e-300)

    # Verify row stochasticity
    assert np.allclose(np.sum(A_new, axis=1), 1, rtol=1e-5), "A matrix not row␣
↪stochastic"
    assert np.allclose(np.sum(B_new, axis=1), 1, rtol=1e-5), "B matrix not row␣
↪stochastic"

    return A_new, B_new
```

```python
def myBW(data, initial_params, itmax):
    """
    Main Baum-Welch algorithm with specified initial parameters

    Args:
        data: T-by-1 observation sequence (1D array)
        initial_params: Dictionary containing initial parameters:
            'w': Initial state distribution (mz,)
            'A': Initial transition matrix (mz-by-mz)
            'B': Initial emission matrix (mz-by-mx)
        itmax: Maximum number of iterations

    Returns:
        A_final: Final transition matrix
        B_final: Final emission matrix
        ll_final: Final log likelihood
    """
    # Ensure data is a 1D array
    data = np.asarray(data).flatten()

    # Extract initial parameters
    w = initial_params["w"]
    A = initial_params["A"]
    B = initial_params["B"]

    for iteration in range(itmax):
        # Compute log likelihood
        alpha = forward_pass(data, w, A, B)
        current_ll = np.log(np.sum(alpha[-1]))

        # Update parameters
        A, B = BW_onestep(data, w, A, B)


    return A, B, current_ll

def myViterbi(data, w, A, B):
    """
    Compute the most likely state sequence using the Viterbi algorithm

    Args:
        data: T-by-1 observation sequence (1D array)
        w: Initial state distribution (mz,)
        A: Transition matrix (mz-by-mz)
        B: Emission matrix (mz-by-mx)

    Returns:
        path: Most likely state sequence
        max_prob: Probability of the most likely path
    """
    # Ensure data is 1D array
    data = np.asarray(data).flatten()

    T = len(data)          # Length of sequence
    mz = len(w)            # Number of hidden states

    # Initialize tables
    V = np.zeros((mz, T))  # Viterbi table
    bp = np.zeros((mz, T), dtype=int)  # Backpointer table

    # Initialize first column of Viterbi table
    V[:, 0] = np.log(w) + np.log(B[:, data[0]])

    # Forward pass: Fill tables
    for t in range(1, T):
        for j in range(mz):
            # Calculate probabilities for all possible previous states
            probs = V[:, t-1] + np.log(A[:, j]) + np.log(B[j, data[t]])
            # Find maximum probability and its index
            V[j, t] = np.max(probs)
            bp[j, t] = np.argmax(probs)

    # Backward pass: Retrieve the most likely path
    path = np.zeros(T, dtype=int)
    # Find the most likely final state
    path[-1] = np.argmax(V[:, -1])
    max_prob = V[path[-1], -1]

    # Backtrack through the sequence
    for t in range(T-2, -1, -1):
        path[t] = bp[path[t+1], t+1]

    return path, max_prob
```

[ ]:
```
# Load HMM data
data_url = 'https://liangfgithub.github.io/Data/coding4_part2_data.txt'  #␣
 ↪Update if necessary
data = pd.read_table(data_url, sep="\s+", header=None).values.flatten() - 1
print("HMM Data Shape:", data.shape)
print("First 5 Observations:\n", data[:5])

# Load true state sequence (for comparison)
Z_url = 'https://liangfgithub.github.io/Data/Coding4_part2_Z.txt'  # Update if␣
 ↪necessary
```

```
Z_true = pd.read_table(Z_url, sep="\s+", header=None).values.flatten() - 1  #␣
↪Assuming states are 1-indexed
Z_true = Z_true[~np.isnan(Z_true)]
print("True State Sequence Shape:", Z_true.shape)
print("First 5 True States:\n", Z_true[:5])

# Initialize parameters
mz = 2  # number of hidden states
mx = 3  # number of observation symbols

w_initial = np.array([0.5, 0.5])
A_initial = np.array([[0.5, 0.5], [0.5, 0.5]])
B_initial = np.array([[1/9, 3/9, 5/9], [1/6, 2/6, 3/6]])

initial_params = {
    'w': w_initial,
    'A': A_initial,
    'B': B_initial
}

# Run Baum-Welch algorithm
itmax = 100
A_final, B_final, ll_final = myBW(data, initial_params, itmax)

print("\nFinal Transition Matrix A:")
print(A_final)
print("\nFinal Emission Matrix B:")
print(B_final)
print("\nFinal Log-Likelihood:")
print(ll_final)
```

```
HMM Data Shape: (200,)
First 5 Observations:
 [1 2 2 2 2]
True State Sequence Shape: (200,)
First 5 True States:
 [0. 0. 0. 0. 0.]

Final Transition Matrix A:
[[0.49793938 0.50206062]
 [0.44883431 0.55116569]]

Final Emission Matrix B:
[[0.22159897 0.20266127 0.57573976]
 [0.34175148 0.17866665 0.47958186]]

Final Log-Likelihood:
-202.3062728417872
```

```
[ ]: # Run Viterbi algorithm
path, max_prob = myViterbi(data, w_initial, A_final, B_final)

print("\nMost Likely State Sequence:")
print(path + 1)

print("\nProbability of the Most Likely Path:")
print(max_prob)

# Compare with true states
matches = (path == Z_true).sum()
total = len(Z_true)
accuracy = matches / total
print(f"\nViterbi Path Accuracy: {accuracy * 100:.2f}%")
```

```
Most Likely State Sequence:
[1 1 1 1 1 1 1 2 1 1 1 1 1 2 2 1 1 1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 1 1 2 1 1
 1 1 1 1 1 1 2 2 1 1 1 1 1 1 2 2 2 1 1 1 1 2 2 2 2 1 1 1 1 1 1 1 1 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1
 1 1 1 2 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
 2 2 2 1 1 1 2 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2 2 2 2 2 1 1 1 2 2 2 1 1 1 1 1
 1 1 1 2 2 2 2 2 1 1 1 1 1 1 1]

Probability of the Most Likely Path:
-318.56245145850596

Viterbi Path Accuracy: 100.00%
```

```
[ ]: # Initialize B uniformly
B_uniform = np.full((mz, mx), 1/3)
initial_params_uniform = {
    'w': w_initial,
    'A': A_initial,
    'B': B_uniform
}

# Run Baum-Welch for 20 iterations
print("\nRunning Baum-Welch with Uniform B for 20 Iterations:")
A_final_20, B_final_20, ll_final_20 = myBW(data, initial_params_uniform,␣
  ↪itmax=20)

print("\nTransition Matrix A after 20 Iterations:")
print(A_final_20)
print("\nEmission Matrix B after 20 Iterations:")
print(B_final_20)
print("\nLog-Likelihood after 20 Iterations:")
print(ll_final_20)

# Run Baum-Welch for 100 iterations
print("\nRunning Baum-Welch with Uniform B for 100 Iterations:")
A_final_100, B_final_100, ll_final_100 = myBW(data, initial_params_uniform,␣
 ↪itmax=100)

print("\nTransition Matrix A after 100 Iterations:")
print(A_final_100)
print("\nEmission Matrix B after 100 Iterations:")
print(B_final_100)
print("\nLog-Likelihood after 100 Iterations:")
print(ll_final_100)


B_uniform[0, :] = np.array([[1/4, 1/2, 1/4]])
B_uniform[1, :] = np.array([[1/4, 1/2, 1/4]])
initial_params_uniform = {
    'w': w_initial,
    'A': A_initial,
    'B': B_uniform
}

# Run Baum-Welch for 20 iterations
print("\nRunning Baum-Welch with Uniform B for 20 Iterations:")
A_final_20, B_final_20, ll_final_20 = myBW(data, initial_params_uniform,␣
 ↪itmax=20)

print("\nTransition Matrix A after 20 Iterations:")
print(A_final_20)
print("\nEmission Matrix B after 20 Iterations:")
print(B_final_20)
print("\nLog-Likelihood after 20 Iterations:")
print(ll_final_20)

# Run Baum-Welch for 100 iterations
print("\nRunning Baum-Welch with Uniform B for 100 Iterations:")
A_final_100, B_final_100, ll_final_100 = myBW(data, initial_params_uniform,␣
 ↪itmax=100)

print("\nTransition Matrix A after 100 Iterations:")
print(A_final_100)
print("\nEmission Matrix B after 100 Iterations:")
print(B_final_100)
print("\nLog-Likelihood after 100 Iterations:")
print(ll_final_100)
```

```
Running Baum-Welch with Uniform B for 20 Iterations:

Transition Matrix A after 20 Iterations:
[[0.5 0.5]
 [0.5 0.5]]

Emission Matrix B after 20 Iterations:
[[0.285 0.19  0.525]
 [0.285 0.19  0.525]]

Log-Likelihood after 20 Iterations:
-202.31544020689537

Running Baum-Welch with Uniform B for 100 Iterations:

Transition Matrix A after 100 Iterations:
[[0.5 0.5]
 [0.5 0.5]]

Emission Matrix B after 100 Iterations:
[[0.285 0.19  0.525]
 [0.285 0.19  0.525]]

Log-Likelihood after 100 Iterations:
-202.31544020689537

Running Baum-Welch with Uniform B for 20 Iterations:

Transition Matrix A after 20 Iterations:
[[0.5 0.5]
 [0.5 0.5]]

Emission Matrix B after 20 Iterations:
[[0.285 0.19  0.525]
 [0.285 0.19  0.525]]

Log-Likelihood after 20 Iterations:
-202.31544020689537

Running Baum-Welch with Uniform B for 100 Iterations:

Transition Matrix A after 100 Iterations:
[[0.5 0.5]
 [0.5 0.5]]

Emission Matrix B after 100 Iterations:
[[0.285 0.19  0.525]
```

```
 [0.285 0.19  0.525]]

Log-Likelihood after 100 Iterations:
-202.31544020689537
```

The algorithm stabilizes after exactly one iteration. This makes sense as algorithm is unable to preference any of the hidden states as their outcomes are identical. This doesn't even require a constant value across the matrix, just as long as the rows of B are identical.