pdf

March 13, 2025

0.1 BAUM WELCH ALGORITHM

[2]:
```python
import numpy as np
from scipy.stats import multivariate_normal


def e_step(X, pi, A, mu, sigma2):
    """E-step: forward-backward message passing"""
    # Messages and sufficient statistics
    N, T, K = X.shape
    M = A.shape[0]
    alpha = np.zeros([N, T, M])  # [N,T,M]
    alpha_sum = np.zeros([N, T])  # [N,T], normalizer for alpha
    beta = np.zeros([N, T, M])  # [N,T,M]
    gamma = np.zeros([N, T, M])  # [N,T,M]
    xi = np.zeros([N, T-1, M, M])  # [N,T-1,M,M]

    # Forward messages
    emission_probabilities = np.stack([multivariate_normal.pdf(X, mean=mu[m],␣
 ↪cov=sigma2[m] * np.eye(K)) for m in range(M)], axis=2)
    # Initialize alpha at t=0
    alpha[:, 0, :] = pi * emission_probabilities[:, 0, :]  # [N,M]
    alpha_sum[:, 0] = np.sum(alpha[:, 0, :], axis=1)  # [N,]
    alpha[:, 0, :] = alpha[:, 0, :] / alpha_sum[:, 0, np.newaxis]  # Normalize

    # Forward pass
    for t in range(1, T):
        for m in range(M):
            alpha[:, t, m] = emission_probabilities[:, t, m] * \
                np.sum(alpha[:, t-1, :] * A[:, m], axis=1)

        alpha_sum[:, t] = np.sum(alpha[:, t, :], axis=1)  # [N,]
        alpha[:, t, :] = alpha[:, t, :] / \
            alpha_sum[:, t, np.newaxis]  # Normalize

    # Backward messages
    # Initialize beta at t=T-1
    beta[:, T-1, :] = 1.0
```

```python
    # Backward pass
    for t in range(T-2, -1, -1):
        for m in range(M):
            for n in range(N):
                beta[n, t, m] = np.sum(
                    A[m, :] * emission_probabilities[n, t+1, :] * beta[n, t+1, :
 ↪])


        beta[:, t, :] = beta[:, t, :] / \
            np.sum(beta[:, t, :], axis=1, keepdims=True)

    # Compute gamma (posterior state probabilities)
    for t in range(T):
        gamma[:, t, :] = alpha[:, t, :] * beta[:, t, :]
        # Normalize gamma
        gamma[:, t, :] = gamma[:, t, :] / \
            np.sum(gamma[:, t, :], axis=1, keepdims=True)

    # Compute xi (posterior transition probabilities)
    for t in range(T-1):
        for n in range(N):
            for i in range(M):
                for j in range(M):
                    xi[n, t, i, j] = alpha[n, t, i] * A[i, j] *␣
 ↪emission_probabilities[n, t+1, j] * beta[n, t+1, j]

            # Normalize xi for each sequence and time step
            xi[n, t] = xi[n, t] / np.sum(xi[n, t])

    # Although some of them will not be used in the M-step, please still
    # return everything as they will be used in test cases
    return alpha, alpha_sum, beta, gamma, xi


def m_step(X, gamma, xi):
    """M-step: MLE"""
    N, T, K = X.shape
    M = gamma.shape[2]

    # Updating initial state distribution pi
    pi = np.sum(gamma[:, 0, :], axis=0) / N

    # Updating transition matrix A
    A = np.sum(xi, axis=(0, 1))  # Sum over N and T
    A /= np.sum(A, axis=1, keepdims=True)  # Normalize rows
```

```
    # Updating emission parameters mu and sigma2
    state = np.sum(gamma, axis=(0, 1))  # Shape: (M,)

    # Update mean vectors mu
    mu = np.einsum('ntm,ntk->mk', gamma, X) / state[:, np.newaxis]

    sigma2 = np.einsum('ntm,ntmk->m', gamma, (X[:, :, np.newaxis, :] - mu[np.
↪newaxis, np.newaxis, :, :]) ** 2) / (state * K)
    return pi, A, mu, sigma2

def hmm_train(X, pi, A, mu, sigma2, em_step=20):
    """Run Baum-Welch algorithm."""
    for step in range(em_step):
        alpha, alpha_sum, beta, gamma, xi = e_step(X, pi, A, mu, sigma2)
        pi, A, mu, sigma2 = m_step(X, gamma, xi)
        print(f"step: {step}  ln p(x): {np.einsum('nt->', np.log(alpha_sum))}")
    return pi, A, mu, sigma2


def hmm_generate_samples(N, T, pi, A, mu, sigma2):
    """Given pi, A, mu, sigma2, generate [N,T,K] samples."""
    M, K = mu.shape
    Y = np.zeros([N, T], dtype=int)
    X = np.zeros([N, T, K], dtype=float)
    for n in range(N):
        Y[n, 0] = np.random.choice(M, p=pi)  # [1,]
        X[n, 0, :] = multivariate_normal.rvs(
            mu[Y[n, 0], :], sigma2[Y[n, 0]] * np.eye(K))  # [K,]
    for t in range(T - 1):
        for n in range(N):
            Y[n, t+1] = np.random.choice(M, p=A[Y[n, t], :])  # [1,]
            # [K,]
            X[n, t+1, :] = multivariate_normal.rvs(
                mu[Y[n, t+1], :], sigma2[Y[n, t+1]] * np.eye(K))
    return X


def main():
    """Run Baum-Welch on a simulated toy problem."""
    # Generate a toy problem
    np.random.seed(12345)  # for reproducibility
    N, T, M, K = 10, 100, 4, 2
    pi = np.array([.0, .0, .0, 1.])  # [M,]
    A = np.array([[.7, .1, .1, .1],
                  [.1, .7, .1, .1],
                  [.1, .1, .7, .1],
```

```
                   [.1, .1, .1, .7]])  # [M,M]
    mu = np.array([[2., 2.],
                    [-2., 2.],
                    [-2., -2.],
                    [2., -2.]])  # [M,K]
    sigma2 = np.array([.2, .4, .6, .8])  # [M,]
    X = hmm_generate_samples(N, T, pi, A, mu, sigma2)

    # Run on the toy problem
    pi_init = np.random.rand(M)
    pi_init = pi_init / pi_init.sum()
    A_init = np.random.rand(M, M)
    A_init = A_init / A_init.sum(axis=-1, keepdims=True)
    mu_init = 2 * np.random.rand(M, K) - 1
    sigma2_init = np.ones(M)

    pi, A, mu, sigma2 = hmm_train(
        X, pi_init, A_init, mu_init, sigma2_init, em_step=20)
    print(pi)
    print(A)
    print(mu)
    print(sigma2)


if __name__ == '__main__':
    main()
```

```
step: 0  ln p(x): -5875.823139965102
step: 1  ln p(x): -4118.059602496055
step: 2  ln p(x): -3857.793423006633
step: 3  ln p(x): -3703.3657441127584
step: 4  ln p(x): -3657.414307811613
step: 5  ln p(x): -3651.2553351711854
step: 6  ln p(x): -3646.6899938211736
step: 7  ln p(x): -3633.535967163485
step: 8  ln p(x): -3584.8088687826885
step: 9  ln p(x): -3402.095600917815
step: 10  ln p(x): -3080.2055319010587
step: 11  ln p(x): -2989.6746438394075
step: 12  ln p(x): -2989.155843425312
step: 13  ln p(x): -2989.154097175903
step: 14  ln p(x): -2989.1540832927926
step: 15  ln p(x): -2989.1540830857157
step: 16  ln p(x): -2989.154083081312
step: 17  ln p(x): -2989.1540830812073
step: 18  ln p(x): -2989.1540830812055
step: 19  ln p(x): -2989.154083081204
[8.72150114e-043 1.00816035e-100 1.73465677e-239 1.00000000e+000]
```

```
[[0.67428047 0.09306092 0.12925046 0.10340816]
 [0.08814086 0.6855656  0.11854267 0.10775087]
 [0.1170085  0.1034202  0.67837916 0.10119214]
 [0.07340673 0.10180477 0.13041155 0.69437695]]
[[-1.99069454 -1.93191607]
 [ 2.00260923  2.02036101]
 [-1.98836479  1.93240146]
 [ 2.00557495 -1.92857525]]
[0.63720789 0.18799243 0.39064707 0.84277227]
```

0.2 D-SEPERATION ALGORITHM

You can copy the code below and run it the check_dsep.py file

[ ]:
```
from collections import deque
class Node(object):
    """
    Node in a directed graph
    """
    def __init__(self, name=""):
        """
        Construct a new node, and initialize the list of parents and children.
        Each parent/child is represented by a (key, value) pair in dictionary,
        where key is the parent/child's name, and value is an Node object.
        Args:
            name: a unique string identifier.
        """
        self.name = name
        self.parents = dict()
        self.children = dict()

    def add_parent(self, parent):
        """
        Args:
            parent: an Node object.
        """
        if not isinstance(parent, Node):
            raise ValueError("Parent must be an instance of Node class.")
        pname = parent.name
        self.parents[pname] = parent

    def add_child(self, child):
        """
        Args:
            child: an Node object.
        """
        if not isinstance(child, Node):
            raise ValueError("Parent must be an instance of Node class.")
```

```python
        cname = child.name
        self.children[cname] = child


class BN(object):
    """
    Bayesian Network
    """
    def __init__(self):
        """
        Initialize the list of nodes in the graph.
        Each node is represented by a (key, value) pair in dictionary,
        where key is the node's name, and value is an Node object
        """
        self.nodes = dict()

    def add_edge(self, edge):
        """
        Add a directed edge to the graph.

        Args:
            edge: a tuple (A, B) representing a directed edge A-->B,
                where A, B are two strings representing the nodes' names
        """
        (pname, cname) = edge

        ## construct a new node if it doesn't exist
        if pname not in self.nodes:
            self.nodes[pname] = Node(name=pname)
        if cname not in self.nodes:
            self.nodes[cname] = Node(name=cname)

        ## add edge
        parent = self.nodes.get(pname)
        child = self.nodes.get(cname)
        parent.add_child(child)
        child.add_parent(parent)

    def print_graph(self):
        """
        Visualize the current graph.
        """
        print("-"*50)
        print("Bayes Network:")
        for nname, node in self.nodes.items():
            print("\tNode " + nname)
            print("\t\tParents: " + str(node.parents.keys()))
```

```
            print("\t\tChildren: " + str(node.children.keys()))
        print("-"*50)

    def is_dsep(self, start, end, observed):
        """
        Check whether start and end are d-separated given observed, using the␣
↪Bayes Ball algorithm.
        Args:
            start: a string, name of the first query node
            end: a string, name of the second query node
            observed: a list of strings, names of the observed nodes.
        Returns:
            True if start and end are d-separated given observed, False␣
↪otherwise.
        """
        if start not in self.nodes or end not in self.nodes:
            raise ValueError("Start or end node not found in the graph.")

        # Convert observed list to a set for faster lookup
        observed_set = set(observed)

        # Queue for BFS: (current_node, direction, is_blocked)
        # direction: 'up' (from child to parent) or 'down' (from parent to␣
↪child)
        queue = deque()
        queue.append((self.nodes[start], 'down', False))

        # Visited set to avoid cycles: (node, direction)
        visited = set()

        while queue:
            current_node, direction, is_blocked = queue.popleft()

            # If we reach the end node and the path is not blocked, return False
            if current_node.name == end and not is_blocked:
                return False

            # Skip if this (node, direction) has already been visited
            if (current_node.name, direction) in visited:
                continue
            visited.add((current_node.name, direction))

            # Handle observed nodes
            if current_node.name in observed_set:
                # If the node is observed, the ball is blocked when coming from␣
↪a parent
                if direction == 'down':
```

```python
                    continue  # Blocked
                elif direction == 'up':
                    # Can go to children
                    for child in current_node.children.values():
                        queue.append((child, 'down', False))
            else:
                # If the node is not observed
                if direction == 'down':
                    # Can go to children
                    for child in current_node.children.values():
                        queue.append((child, 'down', is_blocked))
                    # Can go to parents (reverse direction)
                    for parent in current_node.parents.values():
                        queue.append((parent, 'up', is_blocked))
                elif direction == 'up':
                    # Can go to parents
                    for parent in current_node.parents.values():
                        queue.append((parent, 'up', is_blocked))
                    # Can go to children if not a collider
                    if not is_blocked:
                        for child in current_node.children.values():
                            queue.append((child, 'down', False))

        # If no path reaches the end node, return True (d-separated)
        return True


if __name__ == "__main__":
    # Test the BN class
    bn = BN()
    bn.add_edge(('A', 'B'))
    bn.add_edge(('B', 'C'))
    bn.print_graph()

    print(bn.is_dsep('A', 'C', ['B']))  # True (A and C are d-separated given B)
    print(bn.is_dsep('C', 'A', []))     # False (C and A are not d-separated)
```