Homework9

March 19, 2024

1 Homework

Evelina Teran & Kevin Smith

[ ]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from IPython.display import display_markdown
import platform
import os
```

1.1 Problem 1

[ ]:
```
file_path = os.getcwd() + "/"
if platform.system() == "Windows":
    file_path = file_path.replace("/", "\\")

pb1 = pd.read_csv(file_path + "hmm_pb1.csv", header=None)
pb1 = np.array(pb1).squeeze()
```

[ ]:
```
transition_probs = np.array([[0.95, 0.05], [0.10, 0.90]]) # a values
emission_probs = np.array([[1/6, 1/6, 1/6, 1/6, 1/6, 1/6], [1/10, 1/10, 1/10, 1/
 ↪10, 1/10, 1/2]]) # b values
initial_probs = np.array([0.5, 0.5]) # pi values

# # Observed sequence (dice rolls)
# obs_seq = [1, 3, 5, 2, 4, 6]
```

[ ]:
```
# Define number of hidden states (fair, loaded)
NUM_STATES = 2

# Define the number of time steps (observations) on the shape of pb1
num_time_steps = pb1.shape[0]

# Initialize arrays for storing values
log_vit_probs = np.zeros((NUM_STATES, num_time_steps))
maximizing_states = np.zeros((NUM_STATES, num_time_steps))
most_likely_seq = -1 * np.ones_like(pb1)
```

[ ]:
```
# Initialization of the Viterbi algorithm
# Calculate intial log probs for each state
initial_log_probs = np.log(emission_probs[:, pb1[0] - 1] * initial_probs)
log_vit_probs[:, 0] = initial_log_probs
```

[ ]:
```
# Iterative step of the Viterbi algorithm
for t in range(1, pb1.shape[0]):
    # Calculate log probs of transitioning form previous states to current␣
 ↪states
    transition_log_probs = np.log(transition_probs) + log_vit_probs[:, t-1].
 ↪reshape(-1,1)

    # Calculate log probs of emitting the current observations from each state
    emission_log_probs = np.log(emission_probs[:, pb1[t] - 1])

    # Compute total log probs for each state at current time step t
    total_log_probs = emission_log_probs + np.max(transition_log_probs, axis=0)

    # Update Viterbi matrix with total log probs
    log_vit_probs[:, t] = total_log_probs

    # Store maximizing state for this t
    maximizing_states[:, t] = np.argmax(transition_log_probs, axis=0)

# Find most likely sequence of hidden states
# Initialize last stae in most liekly sequence
most_likely_seq[-1] = np.argmax(log_vit_probs[:, -1])

# Iterate backward through time steps to determine the most likely sequence
for t in range(pb1.shape[0] - 1, 0, -1):
    # Determine the state at the current time step based on the pointer from␣
 ↪the next time step
    most_likely_seq[t - 1] = maximizing_states[most_likely_seq[t], t]

# Display the most likely sequence of hidden states
display_markdown("The most probable sequence of the hidden state is:", raw=True)
print(most_likely_seq + 1)  # Add 1 to each state index for human-readable␣
 ↪output
```

The most probable sequence of the hidden state is:

```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 2 2 2 2 2 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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 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 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 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 1 1 1 1 1 1 1 1
 1 1 1 1 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 2 2
```

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1]
```

[ ]:
```
# Define a function that implements the forward-backward algorithms
def forward_backward(dataset, initial_probs, transition_probs, emission_probs):
    log_alpha = np.zeros((2, dataset.shape[0]))
    log_beta = np.zeros((2, dataset.shape[0]))

    # Compute forward probs
    log_alpha[:, 0] = np.log(emission_probs[:, dataset[0] - 1] * initial_probs)
    log_alpha[:, 0] -= log_alpha[:, 0].sum()

    for t in range(1, dataset.shape[0]):
        alpha_numerator = np.log(emission_probs[:, dataset[t] - 1]) + np.log(np.
 ↪sum(transition_probs * np.exp(log_alpha[:, t - 1].reshape(-1, 1)), axis=0))
        alpha_denominator = np.log(np.sum(np.exp(alpha_numerator)))
        log_alpha[:, t] = alpha_numerator - alpha_denominator

    # Compute backward probs
    log_beta[:, -1] = 0.5

    for t in range(dataset.shape[0] - 2, -1, -1):
        beta_numerator = np.log(np.sum(transition_probs * (np.exp(log_beta[:, t␣
 ↪+ 1]) * emission_probs[:, dataset[t + 1] - 1]), axis=1))
        beta_denominator = np.log(np.sum(np.exp(beta_numerator)))
        log_beta[:, t] = beta_numerator - beta_denominator

    return log_alpha, log_beta
```

[ ]:
```
# Display the result
log_alpha, log_beta = forward_backward(pb1, initial_probs, transition_probs,␣
 ↪emission_probs)
display_markdown("The following shows $\\alpha_{138}^1/\\alpha_{138}^2$.",␣
 ↪raw=True)
print(np.exp(log_alpha[0][138] - log_alpha[1][138]))
display_markdown("The following shows $\\beta_{138}^1/ \\beta_{138}^2$.",␣
 ↪raw=True)
print(np.exp(log_beta[0][138] - log_beta[1][138]))
```

The following shows $\alpha_{138}^1/\alpha_{138}^2$.

```
12.487100265980972
```

The following shows $\beta_{138}^1/\beta_{138}^2$.

```
3.540995425670584
```

1.2 Problem 2

[ ]:
```
pb2 = pd.read_csv(file_path + "hmm_pb2.csv", header=None)
pb2 = np.array(pb1).squeeze()
```

[ ]:
```
# Define a function that implements the forward-backward algorithms
def forward_backward(dataset, initial_probs, transition_probs, emission_probs):
    log_alpha = np.zeros((2, dataset.shape[0]))
    log_beta = np.zeros((2, dataset.shape[0]))

    epsilon = 1e-10  # Small epsilon value to handle zero probabilities and␣
↪avoid division by zero

    # Compute forward probs
    emission_probs_safe = np.maximum(emission_probs[:, dataset[0] - 1], epsilon)
    log_alpha[:, 0] = np.log(emission_probs_safe * initial_probs)
    log_alpha[:, 0] -= log_alpha[:, 0].sum()

    for t in range(1, dataset.shape[0]):
        # Handle zero or negative emission probabilities by adding a small␣
↪epsilon value
        emission_probs_safe = np.maximum(emission_probs[:, dataset[t] - 1],␣
↪epsilon)

        alpha_numerator = np.log(emission_probs_safe) + np.log(np.
↪sum(transition_probs * np.exp(log_alpha[:, t - 1].reshape(-1, 1)), axis=0))
        alpha_denominator = np.log(np.sum(np.exp(alpha_numerator)))
        log_alpha[:, t] = alpha_numerator - alpha_denominator


    # Compute backward probs
    log_beta[:, -1] = 0.5

    for t in range(dataset.shape[0] - 2, -1, -1):
        # Handle zero or negative emission and transition probabilities by␣
↪adding a small epsilon value
        emission_probs_safe = np.maximum(emission_probs[:, dataset[t + 1] - 1],␣
↪epsilon)
        transition_probs_safe = np.maximum(transition_probs, epsilon)

        beta_numerator = np.log(np.sum(transition_probs_safe * (np.
↪exp(log_beta[:, t + 1]) * emission_probs_safe), axis=1))
        beta_denominator = np.log(np.sum(np.exp(beta_numerator)))
        log_beta[:, t] = beta_numerator - beta_denominator

    return log_alpha, log_beta
```

```
[ ]: def baum_welch_algorithm(dataset, initial_probs, transition_probs,␣
 ↪emission_probs, num_iter=10):
    # Initialization
    learned_initial_probs = initial_probs # learned pi
    learned_transition_probs = transition_probs # learned a
    learned_emission_probs = emission_probs # learned b
    epsilon = 1e-10

    # Create one hot encoded dataset for calculation
    encoder = OneHotEncoder(categories = [[1,2,3,4,5,6]], sparse_output=False)
    encoded_dataset = np.transpose(encoder.fit_transform(dataset.reshape(-1,1)))

    for i in range(num_iter):
        # Compute forward and backward probabilities
        alpha, beta = forward_backward(dataset, learned_initial_probs,␣
 ↪learned_transition_probs, learned_emission_probs)

        ## E-step
        # Calculate b_x(t+1)^j
        next_emission_probs = learned_emission_probs[:, np.roll(dataset - 1,␣
 ↪-1)]

        # Handle zero or negative emission probabilities by adding a small␣
 ↪epsilon value
        emission_probs_safe = np.maximum(next_emission_probs, epsilon)

        # Compute xi
        xi_numerator = alpha.reshape(2, 1, -1) * learned_transition_probs.
 ↪reshape(2, 2, 1) * np.roll(
            beta.reshape(1, 2, -1), shift=-1, axis=-1) * emission_probs_safe.
 ↪reshape(1, 2, -1)
        xi_denominator = np.sum(xi_numerator, axis=(0, 1)).reshape(1, 1, -1)
        xi_denominator[xi_denominator == 0] = epsilon # Ensure we do not divide␣
 ↪by zero
        xi = xi_numerator / xi_denominator

        # Check for division by zero in xi_denominator
        assert xi_denominator.min() != 0

        # Calculate gamma (expected number of visits to each state)
        gamma_numerator = alpha * beta
        gamma_denominator = np.sum(gamma_numerator, axis=0).reshape(1, -1)
        gamma_denominator[gamma_denominator == 0] = epsilon  # ensure we don't␣
 ↪divide by zero
        gamma = gamma_numerator / gamma_denominator
```

```
        # Check for division by zero in gamma_denominator
        assert gamma_denominator.min() != 0

        ## M-step
        # Update parameters based on the expected counts
        learned_initial_probs = gamma[:, 0]
        learned_transition_probs = np.sum(xi[:, :, :-1], axis=2) / np.
 ↪sum(gamma[:, :-1], axis=1).reshape(-1, 1)
        learned_emission_probs = np.sum(gamma.reshape(2, 1, -1) *␣
 ↪encoded_dataset.reshape(1, 6, -1), axis=2) / np.sum(gamma, axis=1).
 ↪reshape(-1, 1)

    return learned_initial_probs, learned_transition_probs,␣
 ↪learned_emission_probs
```

[ ]:
```
# Run Baum-Welch algorithm
learned_initial_probs, learned_transition_probs, learned_emission_probs =␣
 ↪baum_welch_algorithm(pb2, initial_probs, transition_probs, emission_probs,␣
 ↪2000)
```

[ ]:
```
# Display the learned parameters
display_markdown("The learned initial probability is:", raw=True)
print(learned_initial_probs)
display_markdown("The learned transition probability is:", raw=True)
print(learned_transition_probs)
display_markdown("The learned emission probability is:", raw=True)
print(learned_emission_probs)
```

The learned initial probability is:

```
[0.28806428 0.71193572]
```

The learned transition probability is:

```
[[0.46355313 0.53577229]
 [0.59087941 0.40972353]]
```

The learned emission probability is:

```
[[2.40936207e-01 8.08309542e-12 8.08184746e-12 6.05410064e-12
  7.51223751e-12 7.59063793e-01]
 [3.37287677e-12 2.26580671e-01 2.26580671e-01 2.76931932e-01
  2.64344117e-01 5.56260842e-03]]
```