MA5710 - Assignment 6

Abhinav T K, MA23M002

November 27, 2023

1. Let $\{Y_n\}, n \geq 0$ be an i.i.d sequence of Bernoulli (p) random variables for some fixed $p \in (0, 1)$. Let $X_n = Y_{n-1} + 2Y_n$ for $n \geq 1$. Prove that X_n is a Markov chain. Figure out the states. Determine initial distribution and transition probabilities.

Solution:
Reference: Introduction to Probability, Dimitri P. Bertsekas and John N. Tsitsiklis
Markov property:
The main assumption in Markov processes is that the transition probability p_{ij} apply whenever state i is visited, no matter what happened in the past, and no matter how state i was reached.
Mathematically,

$$\begin{aligned} P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, ..., X_0 = i_0) &= P(X_{n+1} = j | X_n = i) \\ &= p_{ij} \end{aligned}$$

where, pij is the transition probability.

$$p_{ij} = P(X_{n+1} = j | X_n = i), i, j \in S.$$

for all times n, all states $i, j \in S$, and all possible sequences $i_0, ..., i_{n-1}$ of earlier states. Thus, the probability law of the next state X_{n+1} depends on the past only through the value of the present state X_n. X_n belongs to a finite set **S** of possible states, called the state space. For X_n to be a Markov chain, it should satisfy the Markov property.
We can show that using the fact that Y_n are Bernoulli i.i.d variables. That means each value of Yn is independent of others.

$$X_{n-1} = Y_{n-2} + 2Y_{n-1} \tag{1}$$
$$X_n = Y_{n-1} + 2Y_n \tag{2}$$
$$X_{n+1} = Y_n + 2Y_{n+1} \tag{3}$$
$$\tag{4}$$

Let us assume that the present state is $\mathbf{X_n}$. It means we know the values of $\mathbf{Y_{n-1}}$ and $\mathbf{Y_n}$.
To calculate $\mathbf{X_{n+1}}$ we need to know $\mathbf{Y_n}$ and $\mathbf{Y_{n+1}}$.
$\mathbf{Y_{n+1}}$ is iid and $\mathbf{Y_n}$ is known from $\mathbf{X_n}$. Hence we don't need information of $\mathbf{X_{n-1}}$, $\mathbf{X_{n-2}}$ or previous states to get $\mathbf{X_{n+1}}$.

For states of Xn:
Y_n is a Benoulli random variable. So it can take values 0 or 1 for any $n \geq 0$.
We have $X_n = Y_{n-1} + 2Y_n$.
$Y_{n-1} = 0$ and $Y_n = 0$: $X_n = 0$
$Y_{n-1} = 1$ and $Y_n = 0$: $X_n = 1$
$Y_{n-1} = 0$ and $Y_n = 1$: $X_n = 2$
$Y_{n-1} = 1$ and $Y_n = 1$: $X_n = 3$
So X_n can have states in the set {0, 1, 2, 3}.

Initial Distribution, P(X1):

$X_1 = Y_0 + 2Y_1$
X1 can takes any value in set {0, 1, 2, 3} depending on values of Y_0 and Y_1.
$Y_0 = 0$ and $Y_1 = 0 \Rightarrow X_1 = 0$: $P(X_1) = (1-p)^2$

$Y_0 = 1$ and $Y_1 = 0 \Rightarrow X_1 = 1$: $P(X_1) = p(1-p)$
$Y_0 = 0$ and $Y_1 = 1 \Rightarrow X_1 = 2$: $P(X_1) = p(1-p)$
$Y_0 = 1$ and $Y_1 = 1 \Rightarrow X_1 = 3$: $P(X_1) = p^2$

Transition Probabilities:

For transition probability we have to find $P_{ij} = P(X_{n+1}|X_n)$
Let's check it case by case.

$$\begin{aligned}
\mathbf{P(X_{n+1} = 0|X_n = 0)} &= P(Y_n = 0, Y_{n+1} = 0|Y_{n-1} = 0, Y_n = 0)\\
&= P(Y_{n+1} = 0)\\
&= \mathbf{1 - p}\\
\mathbf{P(X_{n+1} = 1|X_n = 0)} &= P(Y_n = 1, Y_{n+1} = 0|Y_{n-1} = 0, Y_n = 0) \quad \text{(Not possible. Yn cannot be 0 and 1 at the same time.)}\\
&= \mathbf{0}\\
\mathbf{P(X_{n+1} = 2|X_n = 0)} &= P(Y_n = 0, Y_{n+1} = 1|Y_{n-1} = 0, Y_n = 0)\\
&= P(Y_{n+1} = 1)\\
&= \mathbf{p}\\
\mathbf{P(X_{n+1} = 3|X_n = 0)} &= P(Y_n = 1, Y_{n+1} = 1|Y_{n-1} = 0, Y_n = 0) \quad \text{(Not possible. Yn cannot be 0 and 1 at the same time.)}\\
&= \mathbf{0}
\end{aligned}$$

This will the elements of first row of the transition probability matrix.
Similarly we can find the remaining 12 cases.
Hence the **transition probability matrix** is:

$$\begin{bmatrix} 1-p & 0 & p & 0 \\ 1-p & 0 & p & 0 \\ 0 & 1-p & 0 & p \\ 0 & 1-p & 0 & p \end{bmatrix}$$

where each column represent state of X_{n+1} and each row is state of X_n.

---

2. Choose some suitable value of p for Q1 and simulate the above Markov chain in Python/Matlab. Write a general script that evaluates whether a given Markov chain converges to steady-state or not. If it converges for Q1, then print the steady-state probabilities and the no. of steps it took to converge. Write the logic behind your code as well.

Code:

```python
# Python code for Markov Chain simulation and convergence check
import numpy as np

def transition_pMat(p):
    # Transition probability for the given problem
    Pmat = np.array([[1-p, 0, p, 0],
                     [1-p, 0, p, 0],
                     [0, 1-p, 0, p],
                     [0, 1-p, 0, p]])
    return Pmat

#  A general function to simulate Markov chain and checks whether it converges to steadystate or not
def markov_chain_simulation(p, initial_state, iter=1000, tolerance=1e-8):
    # Initialize the transition matrix using input probability p
    transition_pmatrix = transition_pMat(p)
    state_probabilities = np.zeros((iter, 4))      # Initialize the state probabilities
    state_probabilities[0][initial_state] = 1-p          # Setting initial state probability

    # Simulating the Markov chain
    for i in range(1, iter):
        state_probabilities[i] = np.dot(state_probabilities[i - 1],transition_pmatrix)

        # Convergence check
        if np.max(np.abs(state_probabilities[i] - state_probabilities[i - 1])) < tolerance:
            return state_probabilities[:i + 1], i + 1

    return state_probabilities, iter

# Setting the parameters
p_value = 0.7   # p value for the transition matrix
initial_state = 0  # Initial state

# Markov chain simulation and checking for convergence
probability, iterations = markov_chain_simulation(p_value, initial_state)

if iterations < 1000:
    print("Steady-state probabilities for p = {p_value}".format(p_value=p_value))
    print(probability[-1])
    print("The Markov chain converged to steady state in {} iterations.".format(iterations))
else:
    print("The Markov chain could not converge to steady-state for the given iterations.")
```

Output:

Steady-state probabilities for **p = 0.7**
[**0.027 0.063 0.063 0.147**]
The Markov chain converged to steady state in **4 iterations**.

Logic behind code:

Here we set the parameters p of Bernoulli experiment to 0.7 and initial state X1 to 0.
The function **transition_pMat** stores the transition probabilities of X_{n+1} from X_n.
The function **markov_chain_simulation** is a general function to simulate Markov chain and checks whether it converges to steady-state or not. In the simulation we calculate the probaility of each step and checks for convergence if the difference between previous step and current step is less than the tolerance value provided.
Here we do the simulation for 1000 iterations and checks for convergence in the given number of iterations.

3. What is the relation between the Poisson process and the Continuous-time Markov chain? Give an algorithm to simulate a Continuous-time Markov Chain. Use that algorithm to simulate a Poisson process in Python/Matlab. You can choose the parameter values needed in your simulation suitably.

The Poisson process is a specific kind of Continuous-time Markov Chain with continuous state space, where events occur independently at constant rates within non-overlapping intervals.
In a Poisson process:
Continuous State Space: The state space represents time, allowing for events to occur continuously.
Exponential Interarrival Times: The times between successive events, known as interarrival times, follow an exponential distribution. This exponential distribution characterizes the waiting times in a Continuous-time Markov Chain with continuous state space.
Constant Rate Parameter λ: This parameter determines the average rate at which events occur. The higher the λ, the more events occur in a given time frame.

Code:

```python
# Algorithm to simulate a Continuous-time Markov Chain
import numpy as np
def simulate_continuous_time_markov_chain(rate, sim_duration):
    curr_time = 0
    curr_state = 0
    states_list = [curr_state]
    times_list = [curr_time]

    while curr_time < sim_duration:
        time_add = np.random.exponential(scale=1/rate)
        curr_time += time_add

        # Generate a random transition
        transition_prob = np.random.rand()
        if transition_prob < 0.5:
            curr_state = 0 if curr_state == 1 else 1
        states_list.append(curr_state)
        times_list.append(curr_time)
    return states_list, times_list

# Set parameters to simulate Poisson process
transition_rate = 0.4  # Rate parameter
simulation_duration = 10  # Total simulation time

# Simulate the Poisson process
chain_states, chain_times = simulate_continuous_time_markov_chain(transition_rate, simulation_duration)

print("Simulated State:", chain_states)
print("Corresponding Time:", chain_times)
```

Output:
Simulated State: [0, 0, 1, 0, 0, 0, 1, 1, 1, 1]
Corresponding Time: [0, 0.6434596692634292, 1.2563685657536205, 2.6795709580650593, 4.412001079371128, 6.3637051179470
6.411997731712481, 8.238221881858282, 9.136283569449361, 11.774389227485745]

4. Consider the Dataset `https://storage.googleapis.com/download.tensorflow.org/data/iris_training.csv`. This specific dataset separates flowers into three different classes of species; Setosa, Versicolor, and Virginica. The information about each flower contains Sepal length, Sepal width, petal length, and petal width. Taking only Petal length and Petal width as attributes, build the SLIQ classifier and train it on the above dataset.

Code

```
# Python code for SLIQ algorithm
# Here we use the Iris dataset as an example
# The classes are 0: Iris-setosa, 1: Iris-versicolor, 2: Iris-virginica
# The feauters taken are PL: Petal Length, PW: Petal Width
# Model trained using iris_training.csv
# Model tested using iris_test.csv

import csv
# Function to prepare the dataset from the csv file
def data_prep(file_name):
    cols = {'name': ['PL', 'PW'], 'count': 2}     # PL = Petal Length, PW = Petal Width
    data = []
    with open(file_name, 'r', newline='') as csvfile:
        csv_reader = csv.reader(csvfile)
        next(csv_reader)  # Skip the first row
        for row in csv_reader:
            num_row = [float(item) for item in row[2:]]
            num_row[2] = int(num_row[2])
            data.append(num_row)
    return cols, data

# Helper function to create lists of subsets of columns
def create_set(col_set):
    sublists = []
    length = len(col_set)
    for iteration in range(2 ** length):
        cur_set = []
        for digit in range(length):
            if iteration % 2:
                cur_set.append(col_set[digit])
            iteration //= 2
        sublists.append(cur_set)
    return sublists

# Function to sort the column values and class labels
def pre_sort_features(cols, dataset):
    attr_lists = []
    cols['value'] = {}

    for colIdx in range(2):
        cols['value'][colIdx] = sorted(list(set([data[colIdx] for data in dataset])))

        curr_attr_list = []
        for dataIndex in range(len(dataset)):
            curr_attr_list.append([dataset[dataIndex][colIdx], dataIndex])
        attr_lists.append(sorted(curr_attr_list))
    return attr_lists, [[data[cols['count']], 1] for data in dataset]

# Function to calculate the Gini index
def gini_calculator(count_0, count_1, count_2):
    total = count_0 + count_1 + count_2
    if total == 0:
        return 0

    prob_0 = count_0 / total
    prob_1 = count_1 / total
    prob_2 = count_2 / total

    gini = 1 - (prob_0 ** 2 + prob_1 ** 2 + prob_2 ** 2)
    return gini

# Function to check if the data is less than the value
def checker(data, value):
    return data < value

# Function to print the value
```

```
def print_value(name, value):
    return str(name) + ' < ' + str(value)

# Function to display the decision tree
def print_tree(tree, no, deep):
    if tree[no] in [0, 1, 2]:  # Check if the node is a leaf node with class label
        node = f'{no} {tree[no]}'
        print(node)
        return
    left = tree[no][2]
    right = tree[no][3]

    if tree[left] in [0, 1, 2]:  # Check if the left child is a leaf node with class label
        left = tree[left]
    if tree[right] in [0, 1, 2]:  # Check if the right child is a leaf node with class label
        right = tree[right]

    node = f'{no} {tree[no][0]} {tree[no][1]} {left} {right}'
    print(node)
    print_tree(tree, tree[no][2], deep + 1)
    print_tree(tree, tree[no][3], deep + 1)

def is_leaf(classList, node):
    node_class = list(set([x[0] for x in classList if x[1] == node]))
    if len(node_class) != 1:
        return "continue"          # Returns "continue" if it's not a leaf node
    else:
        return node_class[0]    # Returns the class label (0, 1, or 2) if it's a leaf node


def root_split_calculation(cols, attr_lists, class_list, node, used):
    data_num = len(class_list)
    min_gini = 1
    min_index = -1
    min_value = 0
    draw = False

    for col_idx in range(cols['count']):
        if col_idx in used:
            continue
        for value in cols['value'][col_idx]:
            count_0 = count_1 = count_2 = 0
            count_total = 0

            for item in range(data_num):
                if class_list[attr_lists[col_idx][item][1]][1] != node:
                    continue
                if checker(attr_lists[col_idx][item][0], value):
                    if class_list[attr_lists[col_idx][item][1]][0] == 0:
                        count_0 += 1
                    elif class_list[attr_lists[col_idx][item][1]][0] == 1:
                        count_1 += 1
                    else:
                        count_2 += 1
                count_total += 1

            gini_index = gini_calculator(count_0, count_1, count_2)
            if gini_index < min_gini:
                min_gini = gini_index
                min_value = value
                min_index = col_idx

                draw = ((count_0 * count_1 > 0) or (count_0 * count_2 > 0) or (count_1 * count_2 > 0))
                                                                       and (
                    count_0 + count_1 + count_2 == 0)

                if draw:
                    # Resolve draw based on the majority count
                    max_count = max(count_0, count_1, count_2)
                    if max_count == count_0:
                        draw = 0
                    elif max_count == count_1:
                        draw = 1
                    else:
```

```python
                        draw = 2

    return min_index, min_value, draw

def build_tree(cols, col_lists, class_list):
    tree = {}
    tree[1] = 0  # Root node starts with class 0
    queue = [[1]]
    data_num = len(class_list)

    while queue:
        t_node = queue.pop(0)
        node = t_node[0]
        used = t_node[1:]

        flag = is_leaf(class_list, node)
        if flag != "continue":
            tree[node] = flag
            continue

        min_index, min_value, draw = root_split_calculation(cols, col_lists, class_list, node, used)

        if min_index == -1:
            continue

        if draw is not False:
            tree[node] = draw
            continue

        left = len(tree) + 1
        right = left + 1

        tree[node] = [tree[node], print_value(cols['name'][min_index], min_value),
                      left, right, min_index, min_value]

        tree[left] = 0
        tree[right] = 0

        for item in range(data_num):
            if class_list[col_lists[min_index][item][1]][1] != node:
                continue
            if checker(col_lists[min_index][item][0], min_value):
                class_list[col_lists[min_index][item][1]][1] = left
            else:
                class_list[col_lists[min_index][item][1]][1] = right

        q_left = [left]
        q_right = [right]

        for item in used:
            q_left.append(item)
            q_right.append(item)

        if min_index not in used:
            q_left.append(min_index)
            q_right.append(min_index)

        queue.append(q_left)
        queue.append(q_right)

    return tree

def test_decision_node(tree, node, data):
    if tree[node] in [0, 1, 2]:  # Leaf nodes are class labels
        return tree[node]

    attribute_index = tree[node][4]
    attribute_value = tree[node][5]

    if data[attribute_index] < attribute_value:
        return test_decision_node(tree, tree[node][2], data)
    else:
        return test_decision_node(tree, tree[node][3], data)
```

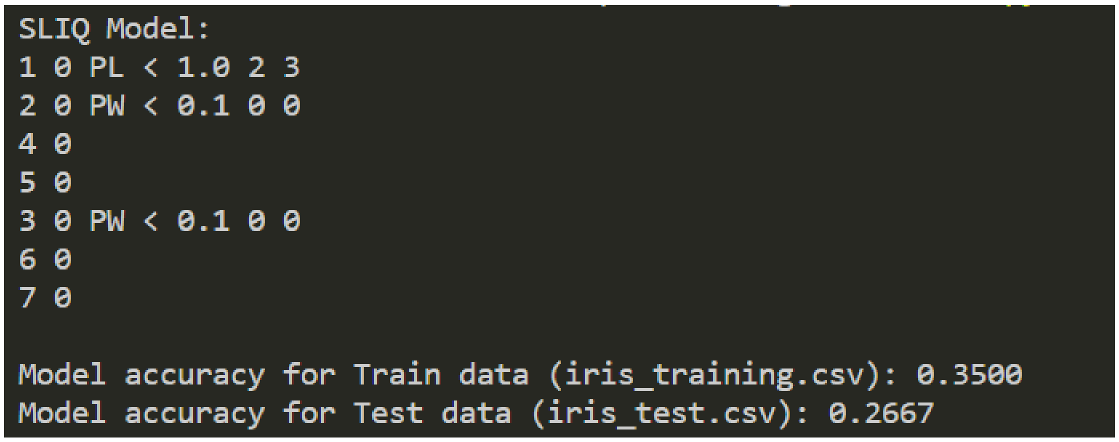
```
def test_decision_tree(tree, test_data):
    length = len(test_data)
    if not length:
        return 'Null'
    count = 0
    for data in test_data:
        if test_decision_node(tree, 1, data) == data[-1]:
            count += 1
    return float(count) / length

def train_decision_tree(file_name):
    columns, train_data = data_prep(file_name)
    col_lists, class_list = pre_sort_features(columns, train_data)
    tree = build_tree(columns, col_lists, class_list)
    print('SLIQ Model:')
    print_tree(tree, 1, 0)
    print('\nModel accuracy for Train data: %.4f' % test_decision_tree(tree, train_data))
    return tree

def decision_tree_testing(file_name, tree):
    _, test_data = data_prep(file_name)
    print('Model accuracy for Test data: %.4f' % test_decision_tree(tree, test_data))

if __name__ == '__main__':
        dtree = train_decision_tree('iris_training.csv')
        decision_tree_testing('iris_test.csv', dtree)
```

Output:



```
SLIQ Model:
1 0 PL < 1.0 2 3
2 0 PW < 0.1 0 0
4 0
5 0
3 0 PW < 0.1 0 0
6 0
7 0

Model accuracy for Train data (iris_training.csv): 0.3500
Model accuracy for Test data (iris_test.csv): 0.2667
```

The model accuracy for train data is 0.35

5. Test the above classifier on the dataset https://storage.googleapis.com/download.tensorflow.org/data/iris_test.csv. Compute the classifier accuracy.
   In the abouve code Q4 we have tested for iris_test.
   Code snippet from above code:

```
def decision_tree_testing(file_name , tree):
    _, test_data = data_prep(file_name)
    print('Model accuracy for Test data (iris_test.csv): %.4f' % test_decision_tree(tree, test_data))

if __name__ == '__main__':
        dtree = train_decision_tree('iris_training.csv')
        decision_tree_testing('iris_test.csv', dtree)
```

   Output:

   The model accuracy for test data is 0.2667

****