EARIN
Laboratory report
EXERCISE 7: Bayes models

Paweł Borsukiewicz & Bartłomiej Mastej

Table of contents

1. Introduction

For the purpose of the laboratory basic inference in Bayesian networks using the MCMC algorithm with Gibbs sampling was performed. Program was written to return the probability distribution of query variables and allows to set the number of steps performed by MCMC algorithm. Estimated data distribution depending on number of steps was illustrated.

2. Theoretical background

A Bayesian network is a directed acyclic graph for which:

- the vertices/nodes correspond to discrete random variables/discrete events,
- the edges represent directly occurring dependencies between these variables,
- for each variable there is a probability distribution defined – this distribution is conditional if there are edges lead into the variable.

Within this exercise 2 nodes are defined – test and cancer. Task explicitly states values of probability of having cancer and conditional probabilities of having cancer depending on the result of the test. Values of probabilities of positive and negative tests can be calculated using following property:

$$P(Cancer = T) = P(Cancer = T|Test = T) * P(Test = T) + P(Cancer = T|Test = F) * P(Test = F) \quad (1)$$

Using equation 1, one may assume that P(Test = T) is equal to x. Therefore, P(Test = F) is equal to 1 – x. Plugging that into the equation the only unknown variable remains x, which as a result can be simply calculated. Knowing values of probabilities of test, one may apply following equation to find conditional probabilities of test depending on cancer:

$$P(Test = T|Cancer = T) = \frac{P(Cancer=T|Test=T) * P(Test=T)}{P(Cancer=T)} \quad (2)$$

Further, one may consider Markov Chain Monte Carlo (MCMC) algorithms. Markov Chain is a model describing a sequence of possible events. Probability of each event within the chain depends only on the state of the previous event. Incorporation of a Markov Chain into Monte Carlo methods results in creation of a random walk. Frequency of visiting particular states during the walk tend to converge to the value of its probability, given the sufficiently high number of steps were simulated.

Gibbs sampling, which is considered to be a special case of the Metropolis–Hastings algorithm, is commonly used for statistical inference and involves choosing a new sample for each dimension separately from the others. In the beginning, we choose randomly values of initial states of all parameters, in our case test and cancer. Then “n” times, where “n” should be sufficiently big number, we determine states of all of the parameters based on the previous state and conditional probabilities. First parameter is chosen entirely on the basis of the values obtained in previous iteration. However, when new sample is chosen for the first of the parameters in a given iteration, it is used later to determine other parameters, rather than the value from the previous state. Similarly, when more than two variables would be considered, the latest values of all parameters would be considered within the conditional probability for j-th variable as presented below:

$$p\left(x_j^{i+1} \middle| x_1^{i+1}, \dots, x_{j-1}^{i+1}, x_{j+1}^{i}, \dots x_d^{i}\right) \quad (3)$$

Where d denotes total number of variables and i is an iteration number. In the end, one may assess the probability of occurrence of all possible states by summing number of occurrences of a given state and dividing it by the number of iterations. Additionally, some number of initial samples may be discarded as they are considered to be a part of a so called burn-in period

3. Implementation

For the purpose of this exercise following set of probabilities was used.

```
def getInitialStates():
    #Test_Cancer
    #We have slightly modified values, because the ones in the task did not add up to 1 properly
    T_T = State(True, True, 0.9)
    T_F = State(True, False, 0.2)
    F_T = State(False, True, 0.1)
    F_F = State(False, False, 0.8)
    return T_T, T_F, F_T, F_F
```

Figure 1. Probabilities

As a result following confusion matrix was expected:

Table 1. Confusion matrix - probabilities

	Test = T	**Test = F**
Cancer = T	0.045	0.005
Cancer = F	0.19	0.76

Our algorithm was implemented within the main function and was composed of two main components – selection of initial values and random walk with “k” iterations selected by the user.

```
def main(k):
    states = createListOfStates()
    probability_of_y = getProbabilityOfY(0.05, states[0], states[1])
    #We start with selecting a ranndom state based on probabilities
    current_state = drawRandomState(states, probability_of_y)
    #Then we save its result
    current_state.update()
    #Further, we perform k steps of random walk
    for _ in range(k):
        current_state = randomWalk(states, current_state, probability_of_y)

    for s in states:
        print("State (Test, Cancer):", s.Y, s.X, "the counter: ", s.counter)

    for s in states:
        print("State (Test, Cancer):", s.Y, s.X, "the probability: ", round(s.counter / (k+1), 5))
```

Figure 2. main() function

In order to draw initial state values we had to calculate value of probability of test being positive.

```python
def getProbabilityOfY(probability_X, state_T_T, state_T_F):
    #Retruns probability of test being positive
    probability_y_true = state_T_T.probability  * probability_X + state_T_F.probability * (1-probability_X)
    return probability_y_true
```

Figure 3. Calculation of probability of test being positive

Having determined that, we moved to the selection of initial state. We have performed it using getState() and drawRandomState() functions.

```python
def getState(states, Y, X):
    #Returns one of four states
    for s in states:
        if s.X == X and s.Y == Y:
            return s
    return None #error - no such state

def drawRandomState(states, p_y):
    #Within the first iteration we want to perform a random choice of the first step
    Y = bernoulli(p_y)
    X = bernoulli(0.05)
    s = getState(states, Y, X)
    return s
```

Figure 4. State selection

Subsequently, we performed random walk using changeState() and randomWalk() functions.

```python
def changeState(states, current_state, change_X, p_y):
    #Get a new state within our random walk
    s = current_state
    if change_X:
        new_X = bernoulli(0.05)
        s = getState(states, current_state.Y, new_X)
    else:
        new_Y = bernoulli(p_y)
        s = getState(states, new_Y, current_state.X)
    return s

def randomWalk(states, current_state, p_y):
    current_state = changeState(states, current_state, bernoulli(0.5), p_y)
    current_state.update()
    return current_state
```

Figure 5. Random walk

Finally, after performing “k” steps of random walk, we printed results to the console.

4. Results

We have performed exemplary tests for 1000 and 100000 steps. Results can be found on figures 6-7, below.

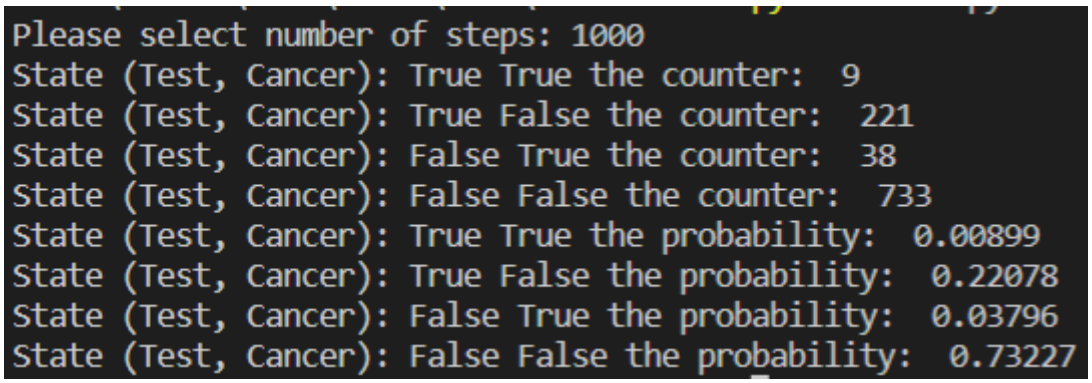

Figure 6. Exemplary results for 1000 steps

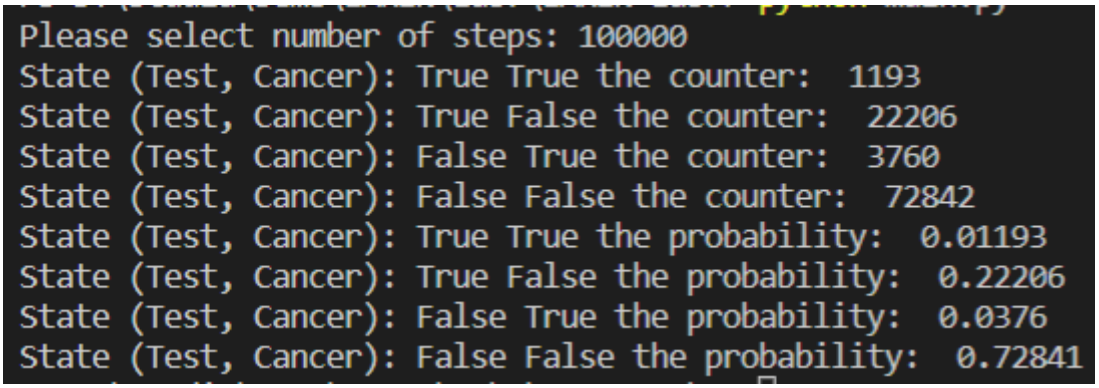

Figure 7. Exemplary results for 100000 steps