UNIVERSITÀ DI PISA

Intelligent Systems For Pattern Recognition

Midterm 2 | Assignment 4

This assignment involves implementing a Bayesian network from scratch.

The network was implemented using 3 classes:

1) ***BayesianNetwork*** - to handle operations on the entire underlying graph.
2) ***BayesianNode*** - to manage the single node.
3) ***CPT_Item*** - an auxiliary class to manage Conditional Probabilities Tables.

Gennaro Daniele **Acciaro**

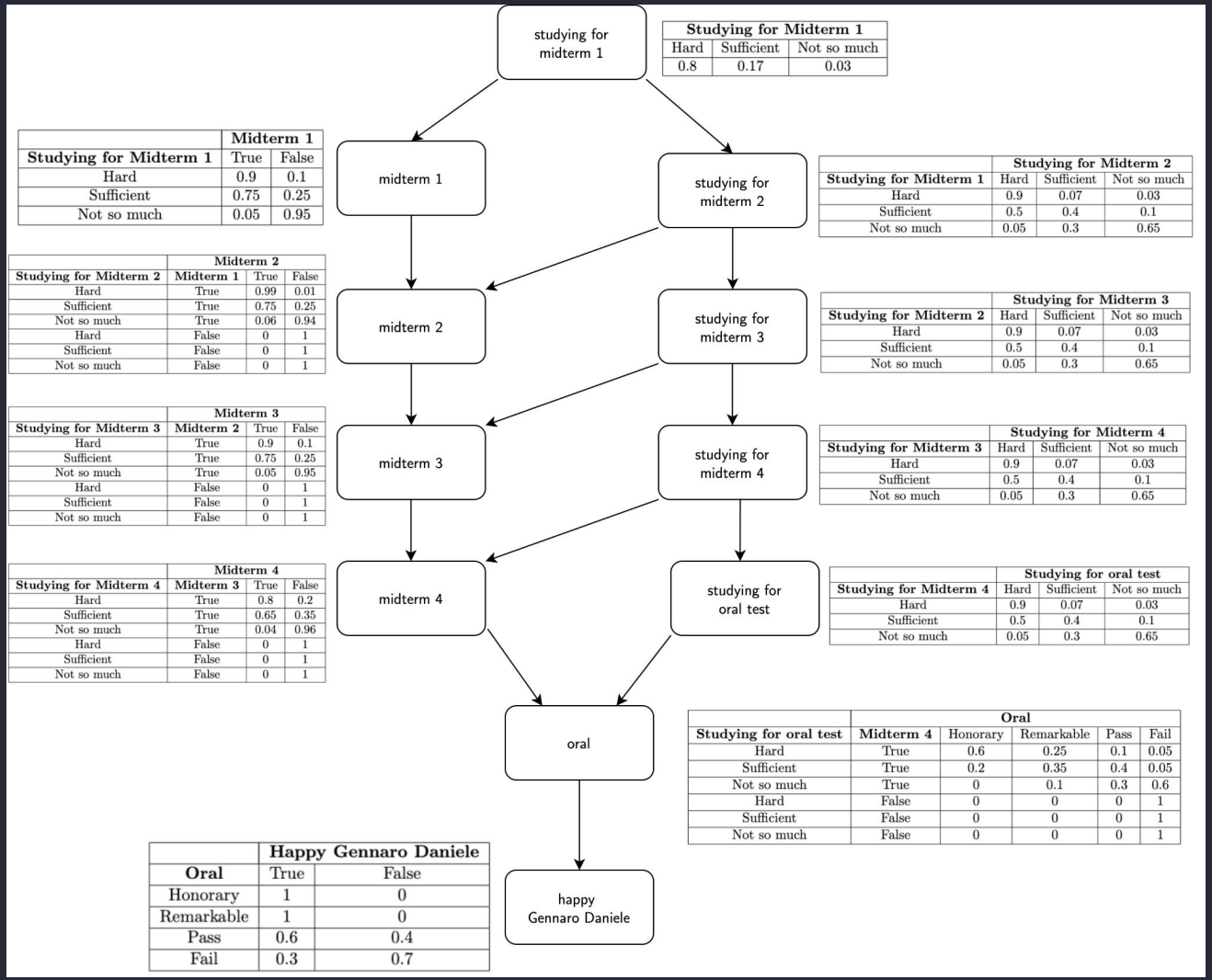
MAT. **635009**

My process

My problem is the path to passing **this exam through the midterms**.

The random variable are: *{studying_m1, studying_m2, studying_m3, studying_m4, studying_oral, midterm1, midterm2, midterm3, midterm4, oral, happy_gennaro_daniele}*

- Each intermediate exam requires the student to have passed the previous midterm and to have studied for that particular midterm. The domain of each midterm exam is boolean.
- Each "studying" node indicates how much a student has studied for a midterm or final oral. The domain of these nodes is ["Hard", "Sufficient", "Not so much"]
- The domain for the oral variable indicates 4 possible grades : ["Honorary", "Remarkable", "Pass", "Fail"]
- The random variable happy_gennaro_daniele indicates my own happiness based on the grade on this exam.



Studying for Midterm 1		
Hard	Sufficient	Not so much
0.8	0.17	0.03

	Midterm 1	
Studying for Midterm 1	True	False
Hard	0.9	0.1
Sufficient	0.75	0.25
Not so much	0.05	0.95

		Studying for Midterm 2		
Studying for Midterm 1		Hard	Sufficient	Not so much
Hard		0.9	0.07	0.03
Sufficient		0.5	0.4	0.1
Not so much		0.05	0.3	0.65

		Midterm 2	
Studying for Midterm 2	**Midterm 1**	True	False
Hard	True	0.99	0.01
Sufficient	True	0.75	0.25
Not so much	True	0.06	0.94
Hard	False	0	1
Sufficient	False	0	1
Not so much	False	0	1

		Studying for Midterm 3		
Studying for Midterm 2		Hard	Sufficient	Not so much
Hard		0.9	0.07	0.03
Sufficient		0.5	0.4	0.1
Not so much		0.05	0.3	0.65

		Midterm 3	
Studying for Midterm 3	**Midterm 2**	True	False
Hard	True	0.9	0.1
Sufficient	True	0.75	0.25
Not so much	True	0.05	0.95
Hard	False	0	1
Sufficient	False	0	1
Not so much	False	0	1

		Studying for Midterm 4		
Studying for Midterm 3		Hard	Sufficient	Not so much
Hard		0.9	0.07	0.03
Sufficient		0.5	0.4	0.1
Not so much		0.05	0.3	0.65

		Midterm 4	
Studying for Midterm 4	**Midterm 3**	True	False
Hard	True	0.8	0.2
Sufficient	True	0.65	0.35
Not so much	True	0.04	0.96
Hard	False	0	1
Sufficient	False	0	1
Not so much	False	0	1

		Studying for oral test		
Studying for Midterm 4		Hard	Sufficient	Not so much
Hard		0.9	0.07	0.03
Sufficient		0.5	0.4	0.1
Not so much		0.05	0.3	0.65

		Oral			
Studying for oral test	**Midterm 4**	Honorary	Remarkable	Pass	Fail
Hard	True	0.6	0.25	0.1	0.05
Sufficient	True	0.2	0.35	0.4	0.05
Not so much	True	0	0.1	0.3	0.6
Hard	False	0	0	0	1
Sufficient	False	0	0	0	1
Not so much	False	0	0	0	1

	Happy Gennaro Daniele	
Oral	True	False
Honorary	1	0
Remarkable	1	0
Pass	0.6	0.4
Fail	0.3	0.7

How I represented CPTs

Each node has a Conditional Probability Table that is implemented as a set of *CPT_Items*, a pair (cause, probability).

Causes are defined by means of a list, and each element in this list contains the cause object and a domain value of this object.

Passing the object directly may not be the most efficient way, computationally speaking, to create CPTs, but I decided to do it this way to make more user-friendly the insertion of values in CPTs. In fact libraries like pomegranate, allow to create CPTs considering using only the domain values. Personally I think this can create confusion and errors if this activity is done manually, even more so if we have to describe a very large CPT.

On the next slide there is a comparison between my version and pomegranate.

Finally, probabilities are handled via a simple list of values. This list follows the order of the domain values of the node itself.

How I represented CPTs

```
    bn.add_probabilities(oral,
        {
            CPT_Item(causes=[[studying_oral, "Hard"], [midterm4, True]],
                     probabilities=[0.6, 0.25, 0.1, 0.05]),
            CPT_Item(causes=[[studying_oral, "Sufficient"], [midterm4, True]],
                     probabilities=[0.2, 0.35, 0.4, 0.05]),
            CPT_Item(causes=[[studying_oral, "Not so much"], [midterm4, True]],
                     probabilities=[0., 0.1, 0.3, 0.6]),

            CPT_Item(causes=[[studying_oral, "Hard"], [midterm4, False]],
                     probabilities=[0, 0, 0, 1]),
            CPT_Item(causes=[[studying_oral, "Sufficient"], [midterm4, False]],
                     probabilities=[0, 0, 0, 1]),
            CPT_Item(causes=[[studying_oral, "Not so much"], [midterm4, False]],
                    probabilities=[0, 0, 0, 1]),
        }

    )
```

My CPT

```
oral = ConditionalProbabilityTable(
    [
        ["Hard", True, "Honorary", 0.6],
        ["Hard", True, "Remarkable", 0.25],
        ["Hard", True, "Pass", 0.1],
        ["Hard", True, "Fail", 0.05],

        ["Sufficient", True, "Honorary", 0.2],
        ["Sufficient", True, "Remarkable", 0.35],
        ["Sufficient", True, "Pass", 0.4],
        ["Sufficient", True, "Fail", 0.05],

        ["Not so much", True, "Honorary", 0.],
        ["Not so much", True, "Remarkable", 0.1],
        ["Not so much", True, "Pass", 0.3],
        ["Not so much", True, "Fail", 0.6],

        ["Hard", False, "Honorary", 0.],
        ["Hard", False, "Remarkable", 0.],
        ["Hard", False, "Pass", 0.],
        ["Hard", False, "Fail", 1.],

        ["Sufficient", False, "Honorary", 0.],
        ["Sufficient", False, "Remarkable", 0.],
        ["Sufficient", False, "Pass", 0.],
        ["Sufficient", False, "Fail", 1.],

        ["Not so much", False, "Honorary", 0.],
        ["Not so much", False, "Remarkable", 0.],
        ["Not so much", False, "Pass", 0.],
        ["Not so much", False, "Fail", 1.],
    ], [studying_oral, midterm4]
)

```

Pomegranate's CPT

How I calculate probabilities

Given a sequence of random variables, we can calculate the probability of the joint distribution of those variables using BN.

In this project, this computation was performed by visiting the graph through the **BFS** graph visitation algorithm. The BFS starts with the independent nodes and adds to the queue all the children of the current node that have not yet been visited.

For each node then, one probability must be selected from its CPT.

- If the node is independent, it will have only one probability in the CPT and then I will use that.
- If the node is dependent, I choose the probability whose causes are present in the causes passed as input to the algorithm.

How I calculate probabilities

```
1     def query(self, param):
2         """ This function performs a BFS on the Bayesian network to find the probabilities to be multiplied.
3             It starts by visiting independent nodes, and then adds their children to the queue.
4             As it visits the queue, it adds the probability given the parents
5             (which have already been visited) to the result.
6          """
7
8         """ Preconditions """
9         if len(param) ≠ len(self.nodes):
10            raise Exception("A query needs every and only the variables in the BN")
11
12        """ The graph visit needs to start from independent nodes """
13        independent_nodes = self.get_independent_nodes()
14
15        """ Once we found independent nodes, we put them to the BFS queue """
16        queue = independent_nodes
17
18        """ Mark all the vertices as not visited """
19        visited = {}
20        for node in self.nodes:
21            visited[node] = False
22
23        prob_found = list()
```

How I

```
24          while queue:
25              item = queue.pop(0) #Remove from head
26
27              """ I execute this for cycle because I don't want
28                      to put constraints in the order of appearance of the random variables in the query """
29              input_value = None
30              for p in param:
31                  input_item = p[0]
32                  value = p[1]
33                  if item == input_item:
34                      input_value = value
35                      break
36
37              value_index = item.domain_values.index(input_value)
38
39              """ safety first """
40              if not hasattr(item, "prob"):
41                  raise Exception("No CPT added to the node <"+str(item)+">")
42
43              for p in item.prob:
44                  selected_probability = p.probabilities[value_index]
45                  if not p.causes:
46                      """If the node is independent"""
47                      prob_found.append(selected_probability)
48                  else:
49                      """If the node is dependent, I choose the probability
50                          whose causes are present in the causes passed as input to the algorithm."""
51                      match_counter = 0
52                      for single_cause in p.causes:
53                          if  single_cause in param:
54                              match_counter+=1
55
56                      if match_counter == len(p.causes):
57                          prob_found.append(selected_probability)
58                          break
59
60              """ Adding my unvisited children to the queue """
61              for child in item.children:
62                  if not visited[child]:
63                      queue.append(child)
64                      visited[child] = True
65
66          result = 1
67          for prob in prob_found:
68              result *= prob
69
70          return result
```

Ancestral Sampling

Ancestral Sampling is a sampling process that involves sampling nodes without parents, and then sampling their children, conditioned by the probability of the parents, and so on.

I implemented this sampling technique through the BFS visitation algorithm.

For each node in the graph I call the ***.sample()*** function which, given a cause, generates a sample for that node given the probability described in the node's CPT for that particular cause.
(For independent nodes, the cause is left blank).

Once the probability is found, I use Numpy's ***np.random.choice()*** method to generate the node's sample (which is a random value from its domain given the probability).

Ancestral Sampling

```python
def ancestral_sampling(self):
    """ The graph visit needs to start from independent nodes """
    independent_nodes = self.get_independent_nodes()

    """ Once we found independent nodes, we put them to the BFS queue """
    queue = independent_nodes

    """ Mark all the vertices as not visited """
    visited = {}
    for node in self.nodes:
        visited[node] = False

    sample_causes = []
    while queue:
        item = queue.pop(0)  # Remove from head
        sample = item.sample(sample_causes)

        sample_causes.append([item, sample])

        """ Adding my unvisited children to the queue """
        for child in item.children:
            if not visited[child]:
                queue.append(child)
                visited[child] = True

    return self.query(sample_causes), sample_causes
```

Bayesian **Network**'s method

Ancestral Sampling

```python
def sample(self, selected_causes = None):
      #Precondition for dependent nodes:
      for p in self.prob:
          if p.causes and selected_causes is None:
              raise Exception("You can't sample the node "+str(self)+" without pass the causes")

      selected_prob = None

      for p in self.prob:
          if p.causes: # If the causes list is not empty

              #The selected_causes input list may contain more causes than are needed for this node
                  #So, I check if all my causes are conteined into the input list, ignoring all the others

              all_causes_are_selected = 0
              for cause in p.causes: # For each my cause ..

                  for single_selected_cause in selected_causes:
                      if cause == single_selected_cause: # .. checks if it's contained
                          all_causes_are_selected+=1

              #If all the causes are contained, I found my probabilities
              if len(p.causes) == all_causes_are_selected:
                  selected_prob = p.probabilities
                  break
          else:
              """ independent nodes """
              selected_prob = p.probabilities
              break

      picked_value = np.random.choice(self.domain_values, 1, p=selected_prob)
      return picked_value[0]
```

Bayesian **Node**'s method

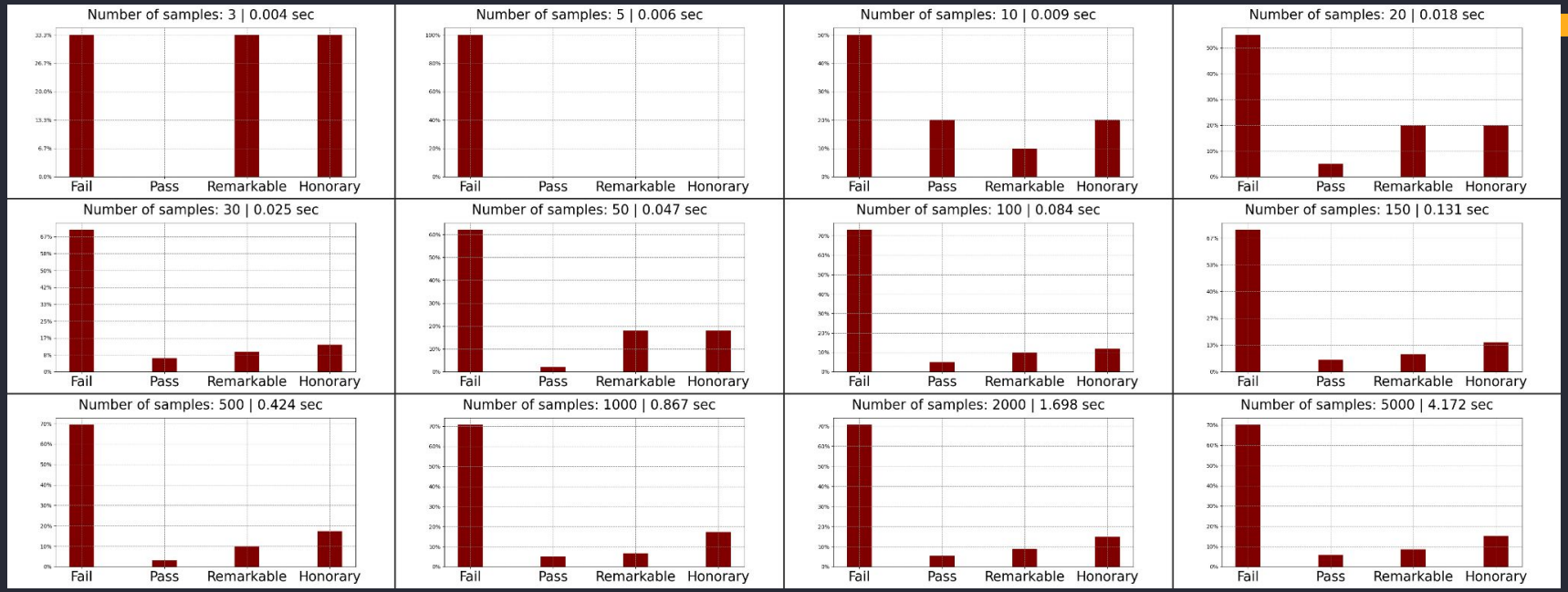
Sampling Results

As we can see from the graphs on the next slide, a fair trade-off between the number of samples (and execution time) and the quality of the distribution is about **100** samples.

In fact the distribution of results with 100 samples is very similar to that with 5000 samples, although the time required for the first run is 0.084 secs while the execution of 2000 samples takes 4.172 secs.

Another interesting observation might be that the distribution follows the idea that, unfortunately, it is much easier to fail this exam (about 70% of total cases) than to pass it with an honors grade (about 15% of total cases).
This was an expected result.

Sampling Results



Finals observations

Building a Bayesian network from scratch was quite a challenge.

Computationally, performing a BFS for each query is expensive, it may be possible to improve the complexity of the project by performing a single visit to the graph after adding all the edges and save this topological order in a data structure and reuse it, whenever it is necessary to visit the graph.

Lessions learned:

- Experience with Bayesian Network
- Experience with Pomegranate (to compare my BN with this library)
- First experience with sampling approaches