Assignment 4

June 24, 2018

Assignment 4
Mahsa Daneshmandmehrabani
V00892119
Logic Programming
Question 1:
Model the following knowledge:
Darth Vader is the parent of Luke Skywalker and Leia Organa.
Leia Organa and Han Solo are the parents of Kylo Ren

```
In [2]: from kanren import Relation, facts

        #Model the following knowledge:
        #Darth Vader is the parent of Luke Skywalker
        #and Leia Organa. Leia Organa and
        #Han Solo are the parents of Kylo Ren.

        parent = Relation()
        facts(parent, ("Darth Vader", "Luke Skywalker"),
                        ("Darth Vader", "Leia Organa"),
                      ("Leia Organa",  "Kylo Ren"),
                      ("Han Solo",  "Kylo Ren"))
```

Question 2:
Formulate the following queries:
who is the parent of Luke Skywalker? who are the children of Darth Vader ?

```
In [3]: #who is the parent of Luke Skywalker

        run(1, x, parent(x, "Luke Skywalker"))

Out[3]: ('Darth Vader',)

In [6]: #who are the children of Darth Vader
        run(2, x, parent("Darth Vader", x))

Out[6]: ('Luke Skywalker', 'Leia Organa')
```

Question 3: Define the relationship grandparent and formulate the query who is the grandparent of Kylo Ren?

```
In [7]: #who is the grandparent of Kylo Ren
        y = var()
        run(1, x, parent(x, y),
                     parent(y, "Kylo Ren"))

Out[7]: ('Darth Vader',)
```

Question 4:
Write functions to answer the queries above using standard Python and not utilizing logic programming.
Answer: I defined the knowledge as a dictionary data structure that the keys are the parents and the values are children. Then I defiend 3 functions: **Who_is_child, Who_is_parent, Who_is_grandparent** to answer the queris above.

```
In [59]: import numpy as np
         #Define the knowledge given in the problem
         #as a dictionary such that the keys are everyone
         #and values are a list of each person's children
         knowledge = {'Darth Vader':['Luke Skywalker', 'Leia Organa'],
                      'Luke':[],
                      'Leia Organa': ['Kylo Ren'],
                      'Han Solo': ['Kylo Ren']}


         def Who_is_child(parent):
             #The value of the parent key
             #in the knowledge dictionary is the child
             return knowledge[parent]


         def Who_is_parent(child):

             parents = []
             #for each key, search if its values contain child return the key
             for i in knowledge:
                 for value in knowledge[i]:
                      if value == child:
                         parents.append(i)
             return parents

         def Who_is_grandparent(child):
             #first get the parents of the child,
             #then get the parents of the parents
             parents = Who_is_parent(child)
             grandparents = []
             for parent in parents:
                 grandparents.append(Who_is_parent(parent))
```

```
            return grandparents



        #who is the parent of Luke Skywalker
        print('who is the parent of Luke Skywalker? ', Who_is_parent('Luke Skywalker'))

        #who is the parent of Luke Skywalker
        print('who is the parent of Kylo Ren? ', Who_is_parent('Kylo Ren'))

        #who is the parent of Luke Skywalker
        print('who are the children of Darth Vader? ', Who_is_child('Darth Vader'))

        ##who is the grandparent of Kylo Ren
        print('who is the grandparent of Kylo Ren? ', Who_is_grandparent('Kylo Ren'))
```

```
who is the parent of Luke Skywalker?  ['Darth Vader']
who is the parent of Kylo Ren?  ['Leia Organa', 'Han Solo']
who are the children of Darth Vader?  ['Luke Skywalker', 'Leia Organa']
who is the grandparent of Kylo Ren?  [['Darth Vader'], []]
```

”Prgramming languages represent computational processes and data structures in programs can store the facts. However, the programming languages lack general mechanism for deriving facts from facts. Each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain. This procedural approach can be contrasted with the declarative nature of propositional logic, in which knowledge and inference are separate, and inference is entirely domain independent. Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.” (from AIMA book)

Hidden Markov Models

Question 1,2:

Random sequences of healthy, and injured states and random sequences of dribble, pass, shoot: First of all, I defined the states, observations, start probabilities, transition probability and emission probabilities. Then I created a **MultinomialHMM** *model* and called the **sample** function on the model that generates 30 random states and observations.

```
In [2]: import numpy as np
        from hmmlearn import hmm

        #Define states
        states = ["Healthy", "Injured"]
        n_states = len(states)

        #Define Observations
        observations = ["Dribble", "Pass", "Shoot"]
        n_observations = len(observations)
```

```
start_probability = np.array([1, 0])

#Define transition probability matrix
transition_probability = np.array([
  [0.7, 0.3],
  [0.5, 0.5]
])


#Define emission probability matrix
emission_probability = np.array([
  [0.2, 0.1, 0.7],
  [0.3, 0.6, 0.1]
])

#Define a HMM model
model = hmm.MultinomialHMM(n_components=n_states)
#Set the start probabilities of the model
model.startprob_=start_probability
#Set the transition probabilities of the model
model.transmat_=transition_probability
#Set the emission probabilities of the model
model.emissionprob_=emission_probability

#O: observations, S: states
#Sample some observations and some
#states
O, S= model.sample(30)

#print(O, S)

#Maping the integer values of O and S to
#names of states and observations
print("States: ", ", ".join(map(lambda y: states[y], S)))


print("Observations: ", ", ".join(map(lambda x: observations[x], O)))
```

```
States:  Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Heal
Observations:  Shoot, Shoot, Shoot, Shoot, Dribble, Shoot, Shoot, Shoot, Shoot, Shoot, Pass, Pa


C:\Users\mahsa\Anaconda3\lib\site-packages\ipykernel_launcher.py:49: VisibleDeprecationWarning
```

Random samples results:

States: Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Injured, Injured, Healthy, Healthy, Healthy, Injured, Healthy, Healthy, Injured, Injured, Healthy, Healthy, Healthy, Healthy, Healthy, Healthy, Injured, Healthy, Healthy

Observations: Shoot, Shoot, Shoot, Shoot, Dribble, Shoot, Shoot, Shoot, Shoot, Shoot, Pass, Pass, Pass, Shoot, Shoot, Dribble, Dribble, Shoot, Shoot, Pass, Dribble, Shoot, Dribble, Shoot, Pass, Shoot, Dribble, Pass, Dribble, Dribble

Question 3, 4:

Use the fit method to learn a Hidden Markov Model for the data you generated.

I generated 300 new states and observations from the model that I defined above. Then I created a new object Model of MultinomialHMM that takes my observations and predicts the transition and emission probabilities.

```
In [32]: #sample 300 observations and states
         #from my model that I defined above
         New_O, New_S= model.sample(300)
         # Define a new MultinomialHMM object that aims to
         #predict the parameters
         new_Model = hmm.MultinomialHMM(n_components=n_states)
         #Calling fit on observations to estimate the
         #parameters
         new_Model.fit(New_O)
         #Model.transmat_ return the Model's transition probability
         print('Transition probabilities: ', new_Model.transmat_)
         #Model.emissionprob_ return the Model's emission probability
         print('Emission probabilities: ', new_Model.emissionprob_)
         #Calling predict functions on observations
         #to predict the states
         PredictedStates = new_Model.predict(New_O)
```

```
Transition probabilities:  [[ 0.72112046  0.27887954]
 [ 0.43737494  0.56262506]]
Emission probabilities:  [[ 0.21256446  0.10192486  0.68551067]
 [ 0.22314841  0.63017193  0.14667965]]
```

As you can see, the predicted transition and emission probabilities are almost close to the actual probabilities.

The number of errors are calculated as follows by deducting the number of matches between original states and predicted ones from 300 samples. The number of matches can be obtained by **np.count_nonzero(PredictedStates == New_S)**.

```
In [35]: #number of errors between states that I generated and the
         #predicted states that predict function generated
         # np.count_nonzero(PredictedStates == New_S) counts the number of matches
         error = 300 - np.count_nonzero(PredictedStates == New_S)
         print('The number of errors is: ', error)
```

```
The number of errors is:  63
```