aima_ch13

October 21, 2020

This is an extract from the official repository of Russell & Norvig's book "Artificial Intelligence - A Modern Approach" (file probability.ipynb in the repository)

1 Probability

This IPy notebook acts as supporting material for topics covered in **Chapter 13 Quantifying Uncertainty** of the book *Artificial Intelligence: A Modern Approach*. This notebook makes use of the implementations in probability.py module. Let us import everything from the probability module. It might be helpful to view the source of some of our implementations. Please refer to the Introductory IPy file for more details on how to do so.

[1]:
```
from probability import *
```

1.1 PROBABILITY DISTRIBUTION

Let us begin by specifying discrete probability distributions. The class **ProbDist** defines a discrete probability distribution. We name our random variable and then assign probabilities to the different values of the random variable. Assigning probabilities to the values works similar to that of using a dictionary with keys being the Value and we assign to it the probability. This is possible because of the magic methods **_ *getitem* _** and **_ *setitem* _** which store the probabilities in the prob dict of the object. You can keep the source window open alongside while playing with the rest of the code to get a better understanding.

[2]:
```
psource(ProbDist)
```

```
<IPython.core.display.HTML object>
```

[3]:
```
p = ProbDist('Flip')
p['H'], p['T'] = 0.25, 0.75
p['T']
```

[3]: 0.75

The first parameter of the constructor **var_name** has a default value of '?'. So if the name is not passed it defaults to ?. The keyword argument **freq** can be a dictionary of values of random variable: probability. These are then normalized such that the probability values sum upto 1 using the **normalize** method.

[4]:
```
p = ProbDist(freq={'low': 125, 'medium': 375, 'high': 500})
p.var_name
```

[4]:
```
'?'
```

[5]:
```
(p['low'], p['medium'], p['high'])
```

[5]:
```
(0.125, 0.375, 0.5)
```

Besides the **prob** and **var_name** the object also separately keeps track of all the values of the distribution in a list called **values**. Every time a new value is assigned a probability it is appended to this list, This is done inside the **_ *setitem* _** method.

[6]:
```
p.values
```

[6]:
```
['low', 'medium', 'high']
```

The distribution by default is not normalized if values are added incrementally. We can still force normalization by invoking the **normalize** method.

[7]:
```
p = ProbDist('Y')
p['Cat'] = 50
p['Dog'] = 114
p['Mice'] = 64
(p['Cat'], p['Dog'], p['Mice'])
```

[7]:
```
(50, 114, 64)
```

[8]:
```
p.normalize()
(p['Cat'], p['Dog'], p['Mice'])
```

[8]:
```
(0.21929824561403508, 0.5, 0.2807017543859649)
```

It is also possible to display the approximate values upto decimals using the **show_approx** method.

[9]:
```
p.show_approx()
```

[9]:
```
'Cat: 0.219, Dog: 0.5, Mice: 0.281'
```

1.2 Joint Probability Distribution

The helper function **event_values** returns a tuple of the values of variables in event. An event is specified by a dict where the keys are the names of variables and the corresponding values are the value of the variable. Variables are specified with a list. The ordering of the returned tuple is same as those of the variables.

Alternatively if the event is specified by a list or tuple of equal length of the variables. Then the events tuple is returned as it is.

[10]:
```
event = {'A': 10, 'B': 9, 'C': 8}
variables = ['C', 'A']
event_values(event, variables)
```

[10]:
```
(8, 10)
```

A probability model is completely determined by the joint distribution for all of the random variables. (**Section 13.3**) The probability module implements these as the class **JointProbDist** which inherits from the **ProbDist** class. This class specifies a discrete probability distribute over a set of variables.

[11]:
```
psource(JointProbDist)
```

```
<IPython.core.display.HTML object>
```

Values for a Joint Distribution is a an ordered tuple in which each item corresponds to the value associate with a particular variable. For Joint Distribution of X, Y where X, Y take integer values this can be something like (18, 19).

To specify a Joint distribution we first need an ordered list of variables.

[12]:
```
variables = ['X', 'Y']
j = JointProbDist(variables)
j
```

[12]:
```
P(['X', 'Y'])
```

Like the **ProbDist** class **JointProbDist** also employes magic methods to assign probability to different values. The probability can be assigned in either of the two formats for all possible values of the distribution. The **event_values** call inside **_ *getitem* _** and **_ *setitem* _** does the required processing to make this work.

[13]:
```
j[1,1] = 0.2
j[dict(X=0, Y=1)] = 0.5

(j[1,1], j[0,1])
```

[13]:
```
(0.2, 0.5)
```

It is also possible to list all the values for a particular variable using the **values** method.

[14]:
```
j.values('X')
```

[14]:
```
[1, 0]
```

1.3 Inference Using Full Joint Distributions

In this section we use Full Joint Distributions to calculate the posterior distribution given some evidence. We represent evidence by using a python dictionary with variables as dict keys and dict

values representing the values.

This is illustrated in **Section 13.3** of the book. The functions **enumerate_joint** and **enumerate_joint_ask** implement this functionality. Under the hood they implement **Equation 13.9** from the book.

$$\mathbf{P}(X|\mathbf{e}) = \alpha\mathbf{P}(X, \mathbf{e}) = \alpha \sum_{y} \mathbf{P}(X, \mathbf{e}, \mathbf{y})$$

Here is the normalizing factor. **X** is our query variable and **e** is the evidence. According to the equation we enumerate on the remaining variables **y** (not in evidence or query variable) i.e. all possible combinations of **y**

We will be using the same example as the book. Let us create the full joint distribution from **Figure 13.3**.

[15]:
```
full_joint = JointProbDist(['Cavity', 'Toothache', 'Catch'])
full_joint[dict(Cavity=True, Toothache=True, Catch=True)] = 0.108
full_joint[dict(Cavity=True, Toothache=True, Catch=False)] = 0.012
full_joint[dict(Cavity=True, Toothache=False, Catch=True)] = 0.016
full_joint[dict(Cavity=True, Toothache=False, Catch=False)] = 0.064
full_joint[dict(Cavity=False, Toothache=True, Catch=True)] = 0.072
full_joint[dict(Cavity=False, Toothache=False, Catch=True)] = 0.144
full_joint[dict(Cavity=False, Toothache=True, Catch=False)] = 0.008
full_joint[dict(Cavity=False, Toothache=False, Catch=False)] = 0.576
```

Let us now look at the **enumerate_joint** function returns the sum of those entries in P consistent with e,provided variables is P's remaining variables (the ones not in e). Here, P refers to the full joint distribution. The function uses a recursive call in its implementation. The first parameter **variables** refers to remaining variables. The function in each recursive call keeps on variable constant while varying others.

[16]:
```
psource(enumerate_joint)
```

```
<IPython.core.display.HTML object>
```

Let us assume we want to find **P(Toothache=True)**. This can be obtained by marginalization (**Equation 13.6**). We can use **enumerate_joint** to solve for this by taking Toothache=True as our evidence. **enumerate_joint** will return the sum of probabilities consistent with evidence i.e. Marginal Probability.

[17]:
```
evidence = dict(Toothache=True)
variables = ['Cavity', 'Catch'] # variables not part of evidence
ans1 = enumerate_joint(variables, evidence, full_joint)
ans1
```

[17]:
```
0.19999999999999998
```

You can verify the result from our definition of the full joint distribution. We can use the same function to find more complex probabilities like **P(Cavity=True and Toothache=True)**

[18]:
```
evidence = dict(Cavity=True, Toothache=True)
variables = ['Catch'] # variables not part of evidence
ans2 = enumerate_joint(variables, evidence, full_joint)
ans2
```

[18]: 0.12

Being able to find sum of probabilities satisfying given evidence allows us to compute conditional probabilities like **P(Cavity=True | Toothache=True)** as we can rewrite this as

$$P(Cavity = True|Toothache = True) = \frac{P(Cavity = True\ and\ Toothache = True)}{P(Toothache = True)}$$

We have already calculated both the numerator and denominator.

[19]:
```
ans2/ans1
```

[19]: 0.6

We might be interested in the probability distribution of a particular variable conditioned on some evidence. This can involve doing calculations like above for each possible value of the variable. This has been implemented slightly differently using normalization in the function **enumerate_joint_ask** which returns a probability distribution over the values of the variable **X**, given the {var:val} observations **e**, in the **JointProbDist P**. The implementation of this function calls **enumerate_joint** for each value of the query variable and passes **extended evidence** with the new evidence having **X = xi**. This is followed by normalization of the obtained distribution.

[20]:
```
psource(enumerate_joint_ask)
```

```
<IPython.core.display.HTML object>
```

Let us find **P(Cavity | Toothache=True)** using **enumerate_joint_ask**.

[21]:
```
query_variable = 'Cavity'
evidence = dict(Toothache=True)
ans = enumerate_joint_ask(query_variable, evidence, full_joint)
(ans[True], ans[False])
```

[21]: (0.6, 0.39999999999999997)

You can verify that the first value is the same as we obtained earlier by manual calculation.