Praktikum 10
Bayesian Networks (1)

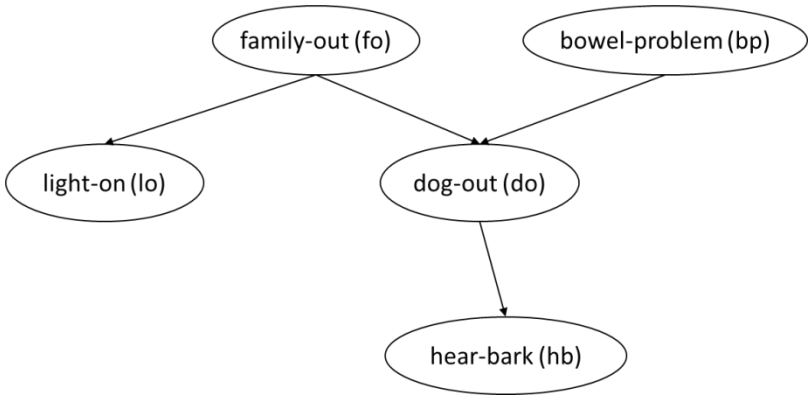
Teorema Bayes

Reverend Thomas Bayes menemukan sebuah teorema yang disebut dengan Teorema Bayes. Teorema Bayes merupakan sebuah teori yang digunakan untuk menghitung probabilitas nilai kebenaran dari suatu *evidence*. Teori ini mengukur derajat kepercayaan subjektif yang berubah secara rasional akibat adanya petunjuk baru. Teori ini sering digunakan dalam ilmu sains, rekayasa, ilmu ekonomi, teori games, kedokteran, serta hukum. Aturan Bayes Rule dapat dilihat pada persamaan 1.

Bayesian Network

Pada Naïve Bayes, metode tersebut mengabaikan korelasi antar variabel, sehingga setiap variabel yang diasumsikan bersifat saling bebas (independence). Sementara itu, Bayesian Network, variabel yang ada bersifat saling dependent / tidak saling bebas / saling mempengaruhi.

Bayesian Network adalah sebuah model peluang dengan menggunakan konsep graf yang merepresentasikan sebuah kumpulan variabel dan keterhubungannya. Bayesian Network adalah sebuah directed acyclic graph (DAG) yang mana setiap node menyatakan variabel-variabel yang digunakan, sedangkan edge menyatakan kondisi bebas bersyarat antara variabel-variabel tersebut. Gambar 1 merupakan contoh dari bayesian network.



Gambar 1. Contoh Bayesian Network

Pada Bayessian networks, kita akan menghitung yakni peluang bersyarat (conditional probabilities) node-node yang diberikan pada network (graph) dimana beberapa nilai yang lain sudah diamati/diketahui.

Probability Distribution

Pada AIMA Python, untuk menghitung probability distribution, kita dapat menggunakan class **ProbDist**. Setiap random variabel yang kita definisikan dan kemudian diberikan nilai probabilitas dengan nilai-nilai yang berbeda dari random variabel tersebut.

```
class ProbDist:

    """A discrete probability distribution.  You name the random variable
    in the constructor, then assign and query probability of values.

    def __init__(self, varname='?', freqs=None):
        """If freqs is given, it is a dictionary of value: frequency pairs,
        and the ProbDist then is normalized."""
        self.prob = {}
        self.varname = varname
        self.values = []
        if freqs:
            for (v, p) in freqs.items():
                self[v] = p
            self.normalize()

    def __getitem__(self, val):
        "Given a value, return P(value)."
        try:
            return self.prob[val]
        except KeyError:
            return 0

    def __setitem__(self, val, p):
        "Set P(val) = p."
        if val not in self.values:
            self.values.append(val)
        self.prob[val] = p

    def normalize(self):
        """Make sure the probabilities of all values sum to 1.
        Returns the normalized distribution.
        Raises a ZeroDivisionError if the sum of the values is 0."""
        total = sum(self.prob.values())
        if not isclose(total, 1.0):
            for val in self.prob:
                self.prob[val] /= total
        return self

    def show_approx(self, numfmt='%.3g'):
        """Show the probabilities rounded and sorted by key, for the
        sake of portable doctests."""
        return ', '.join([('%s: ' + numfmt) % (v, p)
                          for (v, p) in sorted(self.prob.items())])

    def __repr__(self):
        return "P(%s)" % self.varname
```

Misalkan kita membuat sebuah peluang distribusi untuk sebuah random variabel sebagai berikut:

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

Pada potongan kode di atas, kita mendefinisikan sebuah peluang distribusi *p* dari pelemparan sebuah logam (*Flip*). Pada peluang ini terdapat dua variabel yakni kondisi dimana pelemparan menampilkan *Head* atau *Tail*. Di atas, didefinisikan bahwa P('H') = 0.25 dan P('T') = 0.75.

Selain kita dapat bisa mendefinisikan langsung nilai peluangnya, kita juga dapat memberikan nilai frekuensi kejadian untuk setiap variabel. Contoh sebagai berikut:

```
P = ProbDist('X', {'lo': 125, 'med': 375, 'hi': 500})
p.varname
p = ProbDist(freqs={'low': 125, 'medium': 375, 'high': 500})
p.varname
(p['low'], p['medium'], p['high'])
p.values
```

Jika kita tidak mendefiniskan nama variabel dari peluang distribusi, maka default namanya akan diberikan nilai '?'. Potongan kode di atas menyatakan peluang distribusi dari sebuah variabel dengan memberikan nilai frekuensi kejadian pada variabel tersebut. Pada parameter **freqs**, kita menggunakan tipe data dictionary, yakni nama variabel diikuti dengan frekuensi kejadiannya. Kemudian dari nilai frekuensi tersebut, dengan menggunakan method **normalized**, kita akan mendapatkan nilai peluang untuk setiap variabel. Selain bisa mengecek probabilitas dan nama variabel, kita juga dapat mengecek nilai-nilai yang ada pada variabel dengan menggunakan method **values**.

Jika distibusi/frekuensi pada variabel diberikan secara bertahap, maka nilai dari frekuensi tersebut, tidak akan dinormalisasi langsung, kecuali jika kita sudah melakukan proses normalisasi. Kita juga dapat menampilkan nilai frekuensi dengan menggunakan method **show_approx**.

```
p = ProbDist('Y')
p['Cat'] = 50
p['Dog'] = 114
p['Mice'] = 64
(p['Cat'], p['Dog'], p['Mice'])
p.normalize()
(p['Cat'], p['Dog'], p['Mice'])
p.show_approx()
```

Joint Probabilty Distribution

Nilai distribusi probabilitias untuk banyak variabel sekaligus disebut dengan **joint probability distribution**. Contoh: yakni peluang terjadinya dan secara bersaamaan. Kita dapat menggunakan fungsi **event_value** yang mengembalikan tuple dari nilai-nilai variabel pada sebuah event.

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

Untuk mendapatkan probability model maka kita dapat menghitung joint probability untuk semua random variables yang ada. Kita dapat menggunakan class **JointProbDist** yang mewarisi class ProbDist.

```
class JointProbDist(ProbDist):
    """A discrete probability distribute over a set of variables.

    def __init__(self, variables):
        self.prob = {}
        self.variables = variables
        self.vals = defaultdict(list)

    def __getitem__(self, values):
        "Given a tuple or dict of values, return P(values)."
        values = event_values(values, self.variables)
        return ProbDist.__getitem__(self, values)

    def __setitem__(self, values, p):
        """Set P(values) = p.  Values can be a tuple or a dict; it must
        have a value for each of the variables in the joint. Also keep track
        of the values we have seen so far for each variable."""
        values = event_values(values, self.variables)
        self.prob[values] = p
        for var, val in zip(self.variables, values):
            if val not in self.vals[var]:
                self.vals[var].append(val)

    def values(self, var):
        "Return the set of possible values for a variable."
        return self.vals[var]

    def __repr__(self):
        return "P(%s)" % self.variables
```

Sebelum kita menentukan joint distribution, terlebih dahulu kita harus mendefinisilan kumpulan variabel.

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

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

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

j.values('X')
j.values('Y')
```

`Inference menggunakan Full Joint Distribution`

Karena bayesian network dapat memodelkan secara lengkap variabel dan relasi-relasinya. Untuk melengkapinya akan dilakukan perhitungan *full joint distribution*.

Full joint distribution bertujuan untuk menghitung peluang posterior setelah diketahui ***evidence*** / fakta. Pada kode python **probability.py** di aima-python, terdapat kelas **JointProbDist** yang melakukan perhitungan distribusi peluang dari sekumpulan variabel. *Full joint distibution* akan menggunakan kelas **JointProbDist** untuk melakukan perhitungan peluang posterior,dan akan didefinisikan evidence menggunakan tipe data dictionary dengan variabel sebagai *keys* dan nilai peluang sebagai *value*. Contoh dari implementasi ini yaitu sebagai berikut:

```
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
```

Kemudian akan diimplementasikan perhitungan dari Joint Probability dengan membuat fungsi **enumerate_joint** dan **enumerate_joint_ask.** Persamaan untuk menghitung joint probability adalah:

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

Fungsi **enumerate_joint** dibuat untuk melakukan penjumlahan dari semua peluang yang terkait dengan evidence atau yang konsisten terhadap evidence. Misalnya peluang P(X| e), maka akan dihitung penjumlahan dari peluang X dan bukan X. Fungsi dalam python dapat dilihat pada kode dibawah ini:

```
def enumerate_joint(variables, e, P):
    if not variables:
        return P[e]
```

```
    Y, rest = variables[0], variables[1:]
    return sum([enumerate_joint(rest, extend(e, Y, y), P)
                for y in P.values(Y)])
```

Sedangkan fungsi **enumerate_joint_ask** bertujuan untuk melakukan normalisasi dari peluang joint distribution, yaitu dengan menggabungkan peluang joint probability dengan menjumlahkan distribusi peluang joint pada masing-masing variabel yang dilakukan pada fungsi **enumerate_joint.** Fungsi ini yang nantinya akan digunakan untuk menghitung peluang posterior.

```
def enumerate_joint_ask(X, e, P):
    assert X not in e, "Query variable must be distinct from evidence"
    Q = ProbDist(X)  # probability distribution for X, initially empty
    Y = [v for v in P.variables if v != X and v not in e]  # hidden variables.
    for xi in P.values(X):
        Q[xi] = enumerate_joint(Y, extend(e, X, xi), P)
    return Q.normalize()
```

Setelah itu akan diuji perhitungan peluang **JointProbDist** dengan mendefinisikan evidence dan variabel-variabel nya. Misalnya:

```
evidence = dict(Toothache=True)
variables = ['Cavity', 'Catch']
ans1 = enumerate_joint(variables, evidence, full_joint)
```

Kode diatas akan menghitung masing-masing peluang **Cavity** dan **Catch** yang nilainya sudah didefiniskan pada dictionary**,** kemudian akan dihitung total penjumlahan peluang joint masing-masing peluang tersebut terhadap evidence. Kemudian akan dilakukan perhitungan lebih lanjut dengan menggunakan fungsi **enumerate_joint_ask**. Implementasi dari fungsi ini berbeda dengan fungsi **enumerate_joint**, pada fungsi ini dihitung normalisasi peluang joint dari masing-masing variabel yang terkait dengan evidence. Seperti yang sudah dijelaskan diatas, fungsi ini akan menggunakan fungsi **enumerate_joint** untuk menghitung total penjumlahan distribusi peluang. Misalnya kita akan mencari peluang cavity setelah diketahui terjadi Toothache , **P(cavity | Toothache =True)** dengan menggunakan **enumerate_joint_ask.**

```
[in]  : query_variable = 'Cavity'
        evidence = dict(Toothache=True)
        ans = enumerate_joint_ask(query_variable, evidence, full_joint)
        (ans[True], ans[False])
[out] : (0.6, 0.39999999999999997)
```