PDSS_nagyHF_colab_v0 (1)

April 15, 2020

1 A pomegranate python programozási környezet Bayes-hálózatok kezelésére és kiterjesztése döntési hálózatokra

1.1 A gyakorlat célja

A gyakorlat egy átfogó gyakorlati példán keresztül mutatja be egy programozási környezet, a python alapú pomegranate könyvtár felhasználását Bayes-hálózatok létrehozására és hálókban történő következtetésre. Ezen funkciók és a bemutatott minták a következtetés érzékenységét vizsgáló kis házifeladat megoldását segítik.

Ezen felül a gyakorlat bemutatja a Bayes-hálók kibővítéseként előállított döntési hálók alkalmazását is, illetve a valószínűségi hálókból történő mintavételezést és az adatokból történő struktúra tanulást is. Ezen funkciók a kötelező nagy házifeladat és a zárthelyit kiváltó házifeladat megoldását segítik.

1.2 Emlékeztető: Bayes-hálózatok

A feltételes valószínűség definíciója szerint két változó együttes valószínűségét fel lehet bontani az alábbi módon: $P(A, B) = P(A|B)P(B)$. Ezen felbontás tetszőleges számú változóra történő általánosítását **lánc-szabálynak** nevezzük, amely n darab változóra az alábbi módon hajtható végre:

$$P(X_1, X_2, ..., X_n) = P(X_n|X_{n-1}, X_{n-2}, ..., X_1)P(X_{n-1}|X_{n-2}, X_{n-3}, ..., X_1)...P(X_2|X_1)P(X_1)$$

Vagy rövidebben:

$$P(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} P(X_i|X_{i-1}, ..., X_1)$$

Egy modell változóinak együttes valószínűsége ilyen ábrázolásban viszonylag nehezen átlátható, főleg nagyobb méretű modellek esetén. Ebből adódóan érdemes valamilyen gráf alapú reprezentációt bevezetnünk, amely áttekinthető formában mutatja be a modell változói között fennálló kapcsolatokat. Először is feltételezzük, hogy a modell minden A változójára létezik egy olyan $Parents(A)$ változóhalmaz, amely halmazok esetén az alábbi szorzat alakban történő felbontás (faktorizáció) érvényes lesz a modell együttes valószínűségére, n darab változót tartalmazó modell esetére:

$$P(X_1, X_2, ..., X_n) = \prod_{i=1}^{n} P(X_i|Parents(X_i))$$

A gráfos reprezentációban ekkor a $Parents(A)$ halmaz elemeit A változó **szüleinek**, azon X változókat pedig, amelyekre a $A \in Parents(X)$ pedig A **gyerekeinek** jelöljük (ha a szövegkörnyezet megengedi az 1-1 megfeleltetés miatt a valószínűségi változó és csomópont megnevezést ekvivalensként használjuk).

Bayes-háló (Definíció): Azt a körmentes irányított gráfot (Directed Acyclic Graph - DAG), amelynek csomópontjai a modellben található valószínűségi változók, élei pedig a változók közötti függőségi (szülő-gyerek) kapcsolatot reprezentálják úgy, hogy az él a szülőből kiindulva annak gyerekébe mutat, **Bayes-hálónak** nevezzük, ha teljesíti fenti faktorizációs egyenletet és minimális. A gyakorlatban egy Bayes-háló definíciójába a gráf azon kvantitatív annotációját is beleértjük, amely minden csomóponthoz rendel egy feltételes valószínűségi eloszlást, amely a szülők értékei függvényében meghatározza a gyermekcsomópont által reprezentált valószínűségi változó eloszlását.

1.3 Következtetés Bayes-hálókban

Valószínűségi modellek esetén következtetés alatt azt értjük, hogy egy adott változóhalmaz (evidencia) értékének ismeretében kiszámítjuk az ismeretlen értékű valószínűségi változók egy részhalmazának együttes valószínűségét.

Az egzakt és a sztochasztikus (avagy Monte Carlo) következtetési módszerek részletes leírása megtalálható: Stuart Russell és Peter Norvig Mesterséges Intelligencia - Modern megközelítésben című könyvének 14.4. és 14.5. fejezetében.

---

2 A programozási környezet

A gyakorlat során feltételezzük a Python nyelv, és a Google Colab környezet alapszintű ismeretét. Ezeken felül lényegében két eszközt fogunk használni: a pomegranate python könyvtárat a valószínűségi hálók implementációjához, illetve az ez által használt pygraphviz könyvtárat a megalkotott hálók vizualizációjához.

Ezek telepítéséhez futtassa le az alábbi kódblokkot:

[0]:
```
%%capture
!pip install pomegranate
!apt-get install -y graphviz-dev
!pip install pygraphviz
```

[0]:
```
```

Várja meg, amíg a telepítés befejeződik, majd importálja a kezdetben szükséges könyvtárakat az alábbi kódblokk lefuttatásával:

[2]:
```
%matplotlib inline

import matplotlib as mpl
# Just so the generated figures won't look too ugly
mpl.rcParams['figure.figsize'] = [3, 2]
mpl.rcParams['figure.dpi'] = 200

from matplotlib import pyplot as plt

import pomegranate as pg; print('pg: ' + pg.__version__)
import numpy as np; print('np: ' + np.__version__)
```

```
pg: 0.12.2
np: 1.18.2
```

Amennyiben a telepítés során, vagy a későbbiekben hibát észlel, vagy bármilyen egyéb kérdése van, jelezze ezt a tárgy Teams csatornáján vagy konzultációkon.

3 A járművezetői fáradtság modelljének bemutatása

A modern autóiparban egyre gyakrabban alkalmaznak vezetői fáradtságot detektáló rendszereket (Adaptive Driver Assistance Systems, ADAS). A gyakorlat egy ilyen modell implementálásán keresztül fogja bemutatni a valószínűségi következtető rendszerek működését egy programozási környezetben.

Egy kiindulási ADAS modell elérhető a BayesCube rendszer grafikus felhasználói felületén keresztül, illetve egy egyszerűsített modell is, amely a továbbiakban használt.

A változók tekintetében az egyszerűség kedvéért feltételezzük, hogy a rendszer már előzetesen feldolgozott, magas szintű információt kap, így a bemenetek mindegyike diszkrét, többségük bináris.

A modellben bevezetendő valószínűségi változók az alábbiak:

- Az út hossza (**LengthOfDrive**): Bináris változó, értéke igaz, hogyha a vezető által egyhuzamban megtett út hossza egy adott határértéknél magasabb.
- Forgalmi dugó (**TrafficJam**): Igaz, hogyha az út során a sofőr jelentős dugóba kerül.
- Lassult szívverés (**LowHeartRate**): Igaz, hogyha a sofőr pulzusa az alvás során jellemző tartományban van.
- A pulzus mérhető például egy, a kormánykeréken elhelyezett szenzor segítségével.
- Sávtartás (**LaneStability**): Attól függően, hogy a sofőr tartja a sávot, enyhén instabilan vezet, vagy éppen elhagyja a sávot, az értéke lehet *Straight*, *Slalom* vagy *Leaving*.
- Szemek (**Eyes**): Értéke lehet *Open*, *Blink* vagy *Closed* attól függően, hogy a sofőr szeme folyamatosan nyitva van, gyakran pislog, vagy teljesen csukva van.
- A szemek viselkedése egy, a sofőrrel szemben elhelyezett kamera képéből kinyerhető valamilyen képfeldolgozó modell segítségével.
- Fáradtság (**Fatigue**): Igaz, hogyha a vezető fáradtsága annyira magas, hogy nagy valószínűséggel elalszik a volánnál.

Valószínűségi változók lehetséges értékeinek gyakran csak boolean típust használtunk, azonban a keretrendszer sok egyéb típus mellett támogatja a string típusú értékeket is. Ebből adódóan az átláthatóság kedvéért érdemes a lehetséges értékek meghatározásánál a modell bemutatásának végén látható feltételes valószínűségi táblában használt megnevezéseket használni. (Tehát 'Straight', 'Slalom' és 'Leaving' értékek a **LaneStability** változónál, illetve 'Open', 'Blink', 'Close' értékek az **Eyes** esetén)

A továbbakban a rövidség érdekében képletek esetén az alábbi jelölést alkalmazzuk a változókra:

Változó neve	Jelölés
LengthOfDrive	LOD
TrafficJam	TJ
LowHeartRate	LHR
LaneStability	LS
Eyes	E
Fatigue	F

Az oksági relációk elképzelt aszimmetrikus tesztelése alapján adódik, hogy az első két változó (**LengthOfDrive**, **TrafficJam**) a fáradtságnak (**Fatigue**) okai, az utánuk következő három pedig (**LowHeartRate**,**LaneStability**, **Eyes**) következményei. Ebből adódóan a változók együttes valószínűsége felírható az alábbi módon:

$$P(LOD, TJ, LHR, LS, E, F) = P(LOD)P(TJ)P(F|LOD, TJ)P(LHR|F)P(LS|F)P(E|F)$$

A változók valószínűségi eloszlását pedig az alábbi **feltételes valószínűségi táblázat** adja meg:

LOD	$P(LOD)$	TJ	$P(TJ)$	LOD	TJ	F	$P(F\|LOD,TJ)$	F	LHR	$P(LHR\|F)$	F	LS	$P(LS\|F)$	F	E	$P(E\|F)$
$true$	0.1	$true$	0.3	$true$	$true$	$true$	0.65	$true$	$true$	0.8	$true$	$Straight$	0.89	$true$	$Open$	0.1
$false$	0.9	$false$	0.7	$true$	$true$	$false$	0.35	$true$	$false$	0.2	$true$	$Slalom$	0.1	$true$	$Blink$	0.85
				$true$	$false$	$true$	0.45	$false$	$true$	0.3	$true$	$Leaving$	0.01	$true$	$Closed$	0.05
				$true$	$false$	$false$	0.55	$false$	$false$	0.7	$false$	$Straight$	0.99	$false$	$Open$	0.94
				$false$	$true$	$true$	0.55				$false$	$Slalom$	0.01	$false$	$Blink$	0.05
				$false$	$true$	$false$	0.45				$false$	$Leaving$	0	$false$	$Closed$	0.01
				$false$	$false$	$true$	0.35									
				$false$	$false$	$false$	0.65									

4 A Bayes-háló létrehozása

Célunk, hogy a **LengthOfDrive**, **TrafficJam**, **Fatigue**, **LowHeartRate**, **LaneStability** és **Eyes** változókból hozzunk létre egy Bayes-hálót. Ehhez először inicializálja a változók eloszlásait az alábbi kódblokk lefuttatásával:

[0]:
```
from pomegranate import DiscreteDistribution, ConditionalProbabilityTable

lengthOfDrive = DiscreteDistribution({True: 0.1,
                                      False: 0.9})

trafficJam = DiscreteDistribution({True: 0.3,
                                   False: 0.7})

fatigue = ConditionalProbabilityTable([
    [True,  True,  True,  0.65],
    [True,  True,  False, 0.35],
    [True,  False, True,  0.45],
    [True,  False, False, 0.55],
    [False, True,  True,  0.55],
    [False, True,  False, 0.45],
    [False, False, True,  0.35],
    [False, False, False, 0.65],
], [lengthOfDrive, trafficJam])

lowHeartRate = ConditionalProbabilityTable([
    [True,  True,  0.8],
    [True,  False, 0.2],
    [False, True,  0.3],
    [False, False, 0.7],
], [fatigue])

laneStability = ConditionalProbabilityTable([
    [True,  True,  0.8],
    [True,  False, 0.2],
    [False, True,  0.3],
    [False, False, 0.7],
], [fatigue])

eyes = ConditionalProbabilityTable([
    [True,  True,  0.8],
    [True,  False, 0.2],
    [False, True,  0.3],
    [False, False, 0.7],
], [fatigue])
```

Mint látható, a Pomegranate könyvtárban az önálló eloszlások (a kódban *DiscreteDistribution*) meghatározása egy-egy Python dictionary segítségével történik, amelyben a kulcsok a változó lehetséges értékei, a kulcsokhoz tartozó értékek pedig azok valószínűsége egy [0-1] tartományba eső valós szám formájában. Ezzel szemben a feltételes valószínűségek meghatározása a fentebb látható módszerhez hasonlóan egy feltételes valószínűségi táblázat (a kódban *ConditionalProbabilityTable*) segítségével történik, amely tulajdonképpen egy kétdimenziós tömb (tömbök tömbje). Ezen tömb utolsó előtti oszlopa mindig a változó lehetséges értékeit, az utolsó oszlopa pedig a hozzájuk tartozó

valószínűséget tárolja a feltételként jelen lévő változók függvényében. Ezen két oszlopon kívül a tömb elején annyi oszlop van, amennyi szülője van a kérdéses változónak. Végül pedig a változó szüleit (vagyis a valószínűségében feltételként megjelölt változókat) egy külön tömbben adjuk meg referenciaként, olyan sorrendben, amilyen sorrendben az oszlopokat hozzájuk akarjuk rendelni.

Ezt követően definiáljuk az ezen eloszlásokhoz tartozó csomópontokat:

[0]:
```
from pomegranate import State

# All nodes should be initialized with their corresponding distributions.
# From now, the unique name (given here) will identify the variables during␣
↪inference.
lengthOfDriveNode = State(lengthOfDrive, name="LengthOfDrive")
trafficJamNode = State(trafficJam, name="TrafficJam")
fatigueNode = State(fatigue, name="Fatigue")
lowHeartRateNode = State(lowHeartRate, name="LowHeartRate")
laneStabilityNode = State(laneStability, "LaneStability")
eyesNode = State(eyes, "Eyes")
```

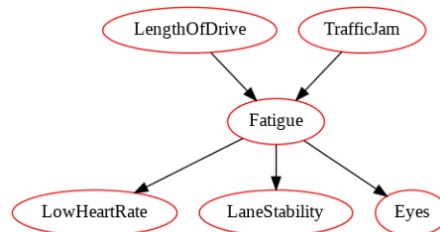
Majd példányosítsunk egy hálót, adjuk hozzá a csomópontokat és a közöttük futó éleket, végül pedig a *bake()* függvény segítségével véglegesítsük a modellt:

[0]:
```
from pomegranate import BayesianNetwork

# Initialize a new Bayesian network
model = BayesianNetwork("Driver Fatigue")

# Add the variables to the model.
# The order only matters if you're planning to index the variables directly␣
↪later.
model.add_states(lengthOfDriveNode,
                 trafficJamNode,
                 fatigueNode,
                 lowHeartRateNode,
                 laneStabilityNode,
                 eyesNode)

# The edges should be added separately here.
# This seems a bit redundant, but note that a distribution can belong to more␣
↪than one node,
# i.e., names in the specification of distributions are not unique identifiers␣
↪(only the ""node identifiers)
model.add_edge(lengthOfDriveNode, fatigueNode)
model.add_edge(trafficJamNode, fatigueNode)
model.add_edge(fatigueNode, lowHeartRateNode)
model.add_edge(fatigueNode, laneStabilityNode)
model.add_edge(fatigueNode, eyesNode)
```

```
# Finally, we must 'bake' the model to finalize it in it's current form.
# We can only perform inference after finalizing the model.
model.bake()
```

Az így elkészült modell struktúráját gráf formájában kirajzolhatjuk a *plot()* függvény segítségével:

[6]:
```
model.plot()
```



5 Következtetés a modellben

A valószínűségi hálókban végrehajtott következtetés során az ismeretlen változók lehetséges értékeinek valószínűségére vagyunk kíváncsiak, a már megfigyelt (és így ismert) változók értékeinek függvényében.

A pomegranate keretrendszerben a *predict_proba()* (azaz *predict probability*) függvény segítségével hajthatunk végre következtetést, amelynél a már ismert változók értékét egy, azok nevével indexelt dictionary-ként adhatjuk meg, visszatérési értékként pedig az összes változó ismert értékét, vagy ismeretlen változók esetén azok aktuális eloszlását kapjuk egy tömb formájában.

A függvény kimenetének áttekinthetőbb kiírásához definiáljunk egy függvényt, az alábbi kódblokk lefuttatásával:

[0]:
```
from pomegranate import Distribution

def print_inference_result(inference_result: []):
  """
    This function prints the result of the 'model.predict_proba' method in a
    more interpretable way than it's original ToString function.
  """
  global model
  assert len(model.states) == len(inference_result)
  for i in range(len(model.states)):
    # We print the name of every variable
    print(model.states[i].name + ':')
    # If a variable is an instance of Distribution,
    # that means that it's value isn't known already.
```

```
    if isinstance(inference_result[i], Distribution):
      # We print out the probability of all possible values of
      # the unknown variable, along with their names.
      for key in inference_result[i].parameters[0].keys():
        print('\t' + str(key) + ':\t' +
              ("%.4f" % inference_result[i].parameters[0][key]))
    else:
      print('\t' + str(inference_result[i]))
    print()
```

5.1 Feltétel nélküli marginálisok kiszámítása

Kezdetben feltételezzük, hogy a változók egyike sem ismert. Ekkor a változók lehetséges értékeinek aktuális valószínűségét egy üres dictionary-vel végzett *predict_proba()* hívással kérdezhetjük le, majd írassuk is ki az eredményt a fenti függvény segítségével:

[8]:
```
evidence_set = {}

inference_result = model.predict_proba(evidence_set)
print_inference_result(inference_result)
```

```
LengthOfDrive:
        True:   0.1000
        False:  0.9000

TrafficJam:
        True:   0.3000
        False:  0.7000

Fatigue:
        False:  0.5800
        True:   0.4200

LowHeartRate:
        False:  0.4900
        True:   0.5100

LaneStability:
        False:  0.4900
        True:   0.5100

Eyes:
        False:  0.4900
        True:   0.5100
```

Az eredményből láthatjuk, hogy amennyiben nincs semmilyen előzetes tudásunk, akkor a fáradtság

valószínűsége 0.42.

##További példák következtetésre

Tegyük fel, hogy a GPS szerint hosszú ideje úton van a sofőr, viszont a tervezett útvonalon nincs forgalmi dugó. Ekkor a hiányzó változók valószínűségét az alábbi módon kaphatjuk meg:

[9]:
```
evidence_set = {"LengthOfDrive": True,
                "TrafficJam":    False}

inference_result = model.predict_proba(evidence_set)
print_inference_result(inference_result)
```

```
LengthOfDrive:
        True

TrafficJam:
        False

Fatigue:
        False:  0.5500
        True:   0.4500

LowHeartRate:
        False:  0.4750
        True:   0.5250

LaneStability:
        False:  0.4750
        True:   0.5250

Eyes:
        False:  0.4750
        True:   0.5250
```

Ezen eredményből megfigyelhetjük, hogy a fáradtság valószínűsége beállt a feltételes valószínűségi táblában a **LengthOfDrive**=True és **TrafficJam**=False evidenciákhoz meghatározott, 0.45-ös értékre. Ez azonban nem jelenti azt, hogy a modell jelenleg ismeretlen változóinak evidenciaként történő bevezetése ne változtathatna a valószínűségén, Bayes-hálókban ugyanis egy változó a szüleinek ismeretében csak a nem-leszármazottaitól válik függetlenné, a maradék három csomópont pedig mind a **Fatigue** változónak leszármazottja.

Feltételezzük, hogy a kormányban elhelyezett szenzor azt jelzi, hogy a sofőr pulzusa normális, nincs közel az alvás során mérhető értékhez. Ekkor a valószínűségek az alábbi módon alakulnak:

[10]:
```
evidence_set = {"LengthOfDrive": True,
                "TrafficJam":    False,
                "LowHeartRate":  False}
```

```
inference_result = model.predict_proba(evidence_set)
print_inference_result(inference_result)
```

```
LengthOfDrive:
        True

TrafficJam:
        False

Fatigue:
        False:  0.8105
        True:   0.1895

LowHeartRate:
        False

LaneStability:
        False:  0.6053
        True:   0.3947

Eyes:
        False:  0.6053
        True:   0.3947
```

Már a feltételes valószínűségi táblában meghatározott értékekből is látható, hogy a **LowHeartRate** viszonylag erősen korrelál a **Fatigue** változóval. Ebből adódóan nem meglepő, hogy a fáradtság valószínűségét ezen változó igazra állítása 0.35-ről ~0.19-re csökkentette, ugyanis az erős evidenciának számít ezen változó False értéke mellett.

Tegyük fel, hogy a vezető arcát néző kamera szerint a sofőr gyakran csukja be a szemét rövidebb időre. Futtassa le az alábbi kódblokkot, amelyben lévő következtetés a meglévő három evidencia mellett már tartalmazza az **Eyes**=‘*Blink*’ evidenciát is:

[11]:
```
evidence_set = {"LengthOfDrive": True,
                "TrafficJam":    False,
                "LowHeartRate":  False,
                "Eyes": 'Blink'}

inference_result = model.predict_proba(evidence_set)
print_inference_result(inference_result)
```

```
LengthOfDrive:
        True

TrafficJam:
        False

Fatigue:
```

```
        False:  0.8105
        True:   0.1895

LowHeartRate:
        False

LaneStability:
        False:  0.6053
        True:   0.3947

Eyes:
        Blink
```

Amennyiben a következtetést megfelelően hajtotta végre, úgy a **Fatigue** változó valószínűségeként ~0.8-at kellett kapjon. Ha figyelembe vesszük, hogy a feltételes valószínűségi tábla szerint fáradtság esetén a gyakori pislogás esélye 0.85, míg fáradtság hiányában csupán ~0.05, akkor láthatjuk, hogy az **Eyes** változó még erősebb korrelációt mutat a fáradtsággal, mint a **LowHeartRate** változó. Ebből adódóan érthető, hogy a fáradtság valószínűségét ezen változó evidenciaként történő bevezetése ~0.19-ről ~0.8-ra növelte.

Végül pedig tegyük fel, hogy az autó külső kamerája szerint a sofőr elkezdte elhagyni a sávot. Vezesse be az alábbi kódblokkban az eddigi evidenciák mellé a **LaneStability**=‘*Leaving*’ értéket, majd futtassa le a blokkot:

[12]:
```python
evidence_set = {"LengthOfDrive": True,
                "TrafficJam":    False,
                "LowHeartRate":  False,
                "Eyes": 'Blink',
                "LaneStability": 'Leaving'}

inference_result = model.predict_proba(evidence_set)
print_inference_result(inference_result)
```

```
LengthOfDrive:
        True

TrafficJam:
        False

Fatigue:
        False:  0.8105
        True:   0.1895

LowHeartRate:
        False

LaneStability:
        Leaving
```

```
Eyes:
        Blink
```

Ha jól hajtotta végre a következtetést, akkor a fáradtság valószínűsége 1. Ez a jelenség viszonylag ritka a valószínűségszámításban, és a valószínűségi következtetésben, és ha be is vezetünk 1 valószínűségű változó-értéket, akkor azt általában evidenciaként szoktuk értelmezni. Ha azonban alaposabban megfigyeljük a feltételes valószínűségi táblázatot, akkor láthatjuk, hogy a **LaneStability**='Leaving' érték valószínűsége 0 abban az esetben, hogyha a fáradtság értéke igaz, és nullánál nagyobb, hogyha nem. Ez röviden azt jelenti, hogy a **LaneStability**='Leaving' érték mellett kizárt, hogy a fáradtság értéke hamis legyen, így tehát mindenképp teljesül, hogy **Fatigue**=True. Lényegében ugyanezt az állítást tükrözi a következtetés során kapott $P(Fatigue = True|LabeStability = Leaving) = 1$ valószínűség is.

6 3. A döntési háló létrehozása

A valószínűségi hálókat leggyakrabban döntéshozó, vagy döntéstámogató rendszerekben szokás használni, ahol egy (vagy több) célváltozó ismert evidenciák melletti valószínűsége alapján szeretnénk meghatározni a lehetséges döntések közül a lehető legjobbat. Ennek végráhajtásához a Bayes-hálót ki kell bővítenünk úgy, hogy meghatározzuk a lehetséges döntéseket, majd hasznosság (vagy veszteség) értékeket rendelünk a célváltozó értékei és a lehetséges döntések összes kombinációjához. Az így kibővített valószínűségi hálót **döntási hálónak** nevezzük.

Döntési hálókban egy adott döntés várható hasznossága (vagy vesztesége) alatt a célváltozó értékeire az adott döntés-érték párhoz meghatározott hasznosság (vagy veszteség) értékét értjük a változóérték aktuális valószínűségével megszorozva, a szorzatot a célváltozó értékei fölött összegezve. Egy adott pillanatban mindig a legnagyobb várható hasznosságú (vagy legkisebb várható veszteségű) döntést érdemes meghozni.

Tegyük fel, hogy kétféle módon tudjuk figyelmeztetni a sofőrt: egy, a kezelőfelületen megjelenő értesítéssel, vagy egy nagy hangerejű figyelmeztető hangjelzéssel. Így tehát minden időpillanatban három lehetséges döntésből kell választanunk: ne csináljunk semmit (*Idle*), jelenítsünk meg egy értesítést a kezelőfelületen (*Notify*), vagy szólaltassunk meg egy hangos figyelmeztetést (*Warning*). Egyértelmű, hogy a legrosszabb eshetőség, hogyha a sofőr elalszik a volánnál, és semmilyen módon nem figyelmeztetjük. Emellett viszont az sem jó, hogyha túlzottan gyakran küldünk figyelmeztetést, ebben az esetben ugyanis a rendszer működése zavaró lehet a sofőr számára, és fennáll az esélye, hogy kikapcsolja azt, ezen felül pedig az is előfordulhat, hogy egy nagy hangerejű hangjelzés megzavarja a sofőrt a vezetésben. Ebből kiindulva vezessük be az alábbi hasznosság értékeket a célváltozó értékeinek, és a döntéseknek lehetséges kombinációira:

$Fatigue$	$Alarm$	$U(Alarm\|Fatigue)$
$true$	$Idle$	-300
$true$	$Notify$	10
$true$	$Warning$	100
$false$	$Idle$	10
$false$	$Notify$	-50

$Fatigue$	$Alarm$	$U(Alarm\|Fatigue)$
$false$	$Warning$	-100

A pomegranate keretrendszer jelen állapotában nem támogatja a döntési hálók létrehozását, így definiálnunk kell egy függvényt, amely meghatározza a lehetséges döntések aktuális hasznosságértékeit a valószínűségi hálón végzett következtetés eredménye alapján. Ezt az alábbi kódblokk lefuttatásával tehetjük meg:

[0]:
```
from pomegranate import Distribution

utilities = {
    (True,  "Idle"):    -300,
    (True,  "Notify"):   10,
    (True,  "Warning"):  100,
    (False, "Idle"):     10,
    (False, "Notify"):  -50,
    (False, "Warning"): -100
}

def print_utility_values(inference_result: []):
  """
    This function computes and prints the utilities of the possible actions,
    based on the probability of the Fatigue variable.
  """
  global model
  assert len(model.states) == len(inference_result)
  expected_utilities = dict()
  for i in range(len(model.states)):
    # We need only the probability of the 'Fatigue' variable
    if model.states[i].name == "Fatigue":
      for key in utilities.keys():
        # Take the probability for the corresponding value of the 'Fatigue'␣
 ↪variable
        prob = inference_result[i].parameters[0][key[0]]
        try:
          # If we've seen this decision before, correct the expected utility.
          expected_utilities[key[1]] += prob * utilities[key]
        except KeyError:
          # Otherwise create a new value for this decision.
          expected_utilities[key[1]] = prob * utilities[key]

      # Get the best decision based on the expected utility values, then print␣
 ↪it.
      best_decision = list(expected_utilities.keys())[0]
      for key in expected_utilities.keys():
        print(key + ": " + str(expected_utilities[key]))
```

```
        if expected_utilities[best_decision] < expected_utilities[key]:
          best_decision = key
      print("The current optimal decision is: " + best_decision)

      return
```

Tegyük fel, hogy nincs forgalmi dugó, nincs hosszú ideje úton a sofőr, és a pulzusa is normális (egy szóval minden rendben van). Ebben a szituációban a lehetséges döntések hasznosságát az alábbi módon írathatjuk ki:

[14]:
```
evidence_set = {"TrafficJam":   False,
                "LengthOfDrive": False,
                "LowHeartRate": False}

inference_result = model.predict_proba(evidence_set)
print_utility_values(inference_result)
```

```
Idle: -31.333333333333492
Notify: -41.99999999999998
Warning: -73.33333333333324
The current optimal decision is: Idle
```

Látható, hogy egy ehhez hasonló, normálisnak mondható szituációban a modell szerint nem érdemes semmilyen figyelmeztetést eszközölni, ugyanis a sofőr nagy valószínűséggel nem álmos.

7 Mintavételezés statisztikai adat generálásához és struktúra tanulás

Alapvetően a valószínűségi hálók a modell változóinak teljes eloszlását hivatottak modellezni, vagy legalábbis közelíteni azt. Ebből adódóan alkalmasak arra is, hogy segítségükkel olyan adatmintákat generáljunk, amelyek valós körülmények között is előfordulhatnak. Más szavakkal élve tehát **mintavételezést** hajtunk végre a modell eloszlása fölött.

A mintavételezési módszerek közül talán a legegyszerűbb a **topologikus sorrendben történő mintavételezés** (*forward sampling*), amely során a háló csomópontjain (ahogyan a név is sejteti) topologikus sorrendben végighaladva mintavételezzük az aktuális csomópont értékét (tehát kvázi sorsolunk neki egy értéket az aktuális eloszlása alapján) a szülei ismeretében.

> A topologikus sorrend ebben az esetben olyan sorrendet jelent, amelynél egy csomópontot csak akkor mintavételezünk, hogyha a szüleinek mindegyikéből vettünk már mintát. Az általunk alkotott Bayes-háló esetén például egy topologikus sorrend az alábbi: *LengthOfDrive, TrafficJam, Fatigue, LowHeartRate, LaneStability, Eyes*
>
> Az egyszerű, topologikus sorrendben történő mintavételezésnél sokoldalúbb és jobb eredményt adó, ám bonyolultabb mintavételezési módszerek is léteznek, ezekről a korábban is említett Mesterséges Intelligencia - Modern megközelítésben című könyv 14.5. fejezetében olvashat.

A mintavételezés mellett egy másik fontos valószínűségi hálókkal kapcsolatos módszer az úgynevezett **struktúra tanulás**. Ennek során lényegében már létező minták sorozatából állítunk elő egy Bayes-hálót, amely lehetőség szerint minél jobban illeszkedik a minták valós eloszlására. Habár a struktúra tanulás elméleti hátterének, és az azt implementáló algoritmusoknak ismertetése bőven túlnyúlik ezen labor határain, a konkrét alkalmazását ettől függetlenül is ki tudjuk próbálni.

A gyorsabb tanulás érdekében a modellbeli paramétereket diszkriminatívabbá tettük.

Sajnos a pomegranate keretrendszer jelenleg nem támogatja natív módon a Bayes-hálókból történő mintavételezést, illetve a struktúra tanulást is csak számszerű (illetve folytonos) értékkészlettel rendelkező valószínűségi változók esetén támogatja közvetlenül. Ezek orvoslása végett a *generate_topological_samples* és *new_model_from_samples* függvényeket az alábbi kódblokk lefuttatásával:

[0]:
```
# The variables are 'quantized', because we gave them numerical values
# instead of the string and boolean values they had before.
# However, the meaning of the possible values are still the same.

lengthOfDrive_quantized = DiscreteDistribution({1: 0.1,
                                                0: 0.9})

trafficJam_quantized = DiscreteDistribution({1: 0.3,
                                             0: 0.7})

fatigue_quantized = ConditionalProbabilityTable([
    [1, 1, 1, 0.65],
    [1, 1, 0, 0.35],
    [1, 0, 1, 0.45],
    [1, 0, 0, 0.55],
    [0, 1, 1, 0.55],
    [0, 1, 0, 0.45],
    [0, 0, 1, 0.35],
    [0, 0, 0, 0.65],
], [lengthOfDrive_quantized, trafficJam_quantized])

lowHeartRate_quantized = ConditionalProbabilityTable([
    [1, 1, 0.8],
    [1, 0, 0.2],
    [0, 1, 0.3],
    [0, 0, 0.7],
], [fatigue_quantized])

laneStability_quantized = ConditionalProbabilityTable([
    [1, 0, 0.2],
    [1, 1, 0.65],
    [1, 2, 0.15],
    [0, 0, 0.9],
    [0, 1, 0.09],
    [0, 2, 0.01]
```

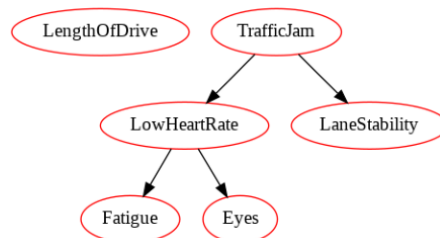
```
], [fatigue_quantized])

eyes_quantized = ConditionalProbabilityTable([
    [1, 0, 0.1],
    [1, 1, 0.85],
    [1, 2, 0.05],
    [0, 0, 0.94],
    [0, 1, 0.05],
    [0, 2, 0.01]
], [fatigue_quantized])

def generate_topological_samples(sample_count: int = 1) -> np.ndarray:
  """
    This function implements the topological sampling algorithm on our model.
  """
  global lengthOfDrive_quantized, trafficJam_quantized, fatigue_quantized
  global lowHeartRate_quantized, laneStability_quantized, eyes_quantized

  result = []
  # We run the sampling 'sample_count' times...
  for i in range(sample_count):
    # First, we sample the variables without parents:
    LOD_sample = lengthOfDrive_quantized.sample()
    TJ_sample = trafficJam_quantized.sample()

    # All of Fatigue's parents are sampled, so we sample Fatigue:
    F_sample = fatigue_quantized.sample(parent_values={lengthOfDrive_quantized:␣
↪LOD_sample,
                                                       trafficJam_quantized:␣
↪TJ_sample})

    # Fatigue is the only parent of the last three variables,
    # so we can sample them in any order by now:
    LHR_sample = lowHeartRate_quantized.sample(parent_values={fatigue_quantized:
↪ F_sample})
    LS_sample = laneStability_quantized.sample(parent_values={fatigue_quantized:
↪ F_sample})
    E_sample = eyes_quantized.sample(parent_values={fatigue_quantized:␣
↪F_sample})

    # Finally, we store the generated sample in the result variable:
    result.append([LOD_sample, TJ_sample, F_sample,
                   LHR_sample, LS_sample, E_sample])

  return np.array(result)

def new_model_from_samples(samples: np.ndarray) -> BayesianNetwork:
```

```
    """
      In this function we basically generate a Bayesian network from the given
      samples with the built-in function 'BayesianNetwork.from_samples', and then
      rename the states to make the resulting network easier to interpret.
    """
    learned_model = BayesianNetwork.from_samples(samples)
    state_names = ["LengthOfDrive",
                   "TrafficJam",
                   "Fatigue",
                   "LowHeartRate",
                   "LaneStability",
                   "Eyes"]
    for i in range(len(learned_model.states)):
      learned_model.states[i].name = state_names[i]
    return learned_model
```

Kezdetben generáljunk 10 darab mintát, majd struktúra tanulással hozzunk létre egy hálót a generált minták alapján az alábbi kódblokk lefuttatásával:

(Itt fontos realizálni, hogy a struktúra tanulás során a modell semmilyen információval nem rendelkezik az eredeti modellről, csak az abból generált mintákat látja)

[25]:
```
num_samples = 10

samples = generate_topological_samples(num_samples)
learned_model = new_model_from_samples(samples)
learned_model.plot()
```



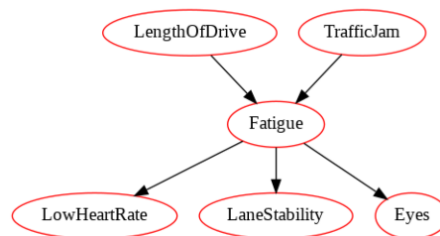
Jól látható, hogy a kialakult modell viszonylag kis hasonlóságot mutat az eredeti valószínűségi hálóval, amelyből a mintákat generáltuk. Ez első sorban annak köszönhető, hogy 10 darab minta jellemzően kevés egy ehhez hasonló, bonyolultabb struktúra felismeréséhez.

Az alábbi kódban a minták számának növelésével kísérletezze ki, hogy nagyjából mekkora mintahalmaz szükséges ahhoz, hogy a kialakult Bayes-háló topológiája hasonlítson az eredetihez:

> Mielőtt belekezdene, mindenképp vegye figyelembe, hogy túlzottan nagy mintaszám esetén a mintagenerálás és a struktúra tanulás számításigénye, és így időtartama is jelentősen megnőhet, így **törekedjen a mintaszám óvatos növelésére**.

Ezen felül érdemes még megjegyezni, hogy ugyanazon mintaszám mellett az egyes futások eredménye drasztikusan eltérő lehet. Ez főként a minták véletlenszerű generálásából adódik, azonban a struktúra tanulás nemdeterminisztikus tulajdonsága is közrejátszik benne. Ebből adódóan **ugyanazon mintaszám mellett érdemes többször is lefuttatni** a blokkot, ugyanis előfordulhat, hogy egy futás így is jobb eredményt ad az előzőnél.

[26]:
```
# TODO: Define a sample count, that is necessary to learn the model accurately.

num_samples = 100000

samples = generate_topological_samples(num_samples)
learned_model = new_model_from_samples(samples)
learned_model.plot()
```



Nagyjából akkor számít jónak az eredmény, hogyha a tanult modell irányítatlan váza megegyezik az eredetivel. Tökéletes az eredmény, ha a tanult modell és az eredeti modell ugyanabba a megfigyelési ekvivalenciaosztályba tartozik. Emlékezzünk vissza, hogy ez megengedi úgynevezett nem kényszerített (non compelled) élek különböző irányítottságát.

Mindemellett az is könnyen előfordulhat, hogy egy változó szülei, vagy gyerekei között valamilyen kapcsolatot vél felfedezni. Ez nem meglepő, ugyanis egy adott változó szülei a kérdéses változón keresztül közvetetten függenek egymástól, illetve hogyha a változó értéke nem ismert, akkor ez a gyerekeire is igaz.