EECS 491 Final Project: Bayesian Structure and Parameter Learning

Overview

Bayesian-network models are a commonly-used framework for modeling conditional probabilities, allowing extremely complicated conditional relationships to be modeled and implemented with ease. Previously in this course, we have covered in great detail how to perform inference operations given a defined bayesian-network model. This works well when we are able to construct an informed model, which includes the correct conditional probabilities and links. However, what are we to do when we are given lots of data, with no knowledge of the relationships between any of the variables or elements? Before we can perform any kind of predictive inference, we first need to learn a bayesian model from our data. This can generally be described in two parts: learning the structural relationship between probabilistic variables, and learning the conditional probability distributions which fit data to a given structure.

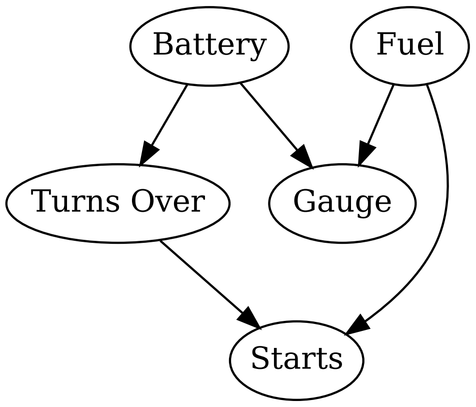
In [90]:
```
from graphviz import Graph, Digraph
import pgmpy
from pgmpy.models import BayesianModel
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination
from pgmpy.sampling import GibbsSampling
from pgmpy.estimators import ExhaustiveSearch, HillClimbSearch, ConstraintBased
Estimator
from pgmpy.estimators import BDeuScore, K2Score
import time
```

True Model

In order to generate bayesian-networks from scratch, we need data to describe. Throughout this example, we will be using the following model (Adpted from Barber Exercise 3.6) that describes the probability of a car starting given certain factors. We will use this model to generate samples which will be used to generate networks and parameters, and then we will compare the accuracy of the generated models to the actual model.

In [2]:
```
q5 = Digraph()
q5.node('b', 'Battery')
q5.node('g', 'Gauge')
q5.node('f', 'Fuel')
q5.node('t', 'Turns Over')
q5.node('s', 'Starts')
q5.edges(['bg','fg','bt','ts','fs'])
q5
```

Out[2]:



In [91]:
```
# Create model from list of edges
model = BayesianModel([('B','G'),('F','G'),('B','T'),('T','S'),('F','S')])

# define p(B) and p(M)
# variable_card is cardinality = 2 for true|false
# values are defined in numeric order p(x_i = [false, true]), ie  [0, 1]
priorB = TabularCPD(variable='B', variable_card=2, values=[[0.02, 0.98]])
priorF = TabularCPD(variable='F', variable_card=2, values=[[0.05, 0.95]])

# define p(G|B,F)
# Variables cycle in numerical order of evidence values,
# ie BF = 00, 01, 10, 11 for each value of G.
cpdG = TabularCPD(variable='G', variable_card=2,
                  evidence=['B', 'F'], evidence_card=[2, 2],
                  values=[[0.99, 0.1, 0.97, 0.04],
                          [0.01, 0.9, 0.03, 0.96]])

# define p(T|B)
cpdT = TabularCPD(variable='T', variable_card=2,
                 evidence=['B'], evidence_card=[2],
                 values=[[0.98, 0.03],
                         [0.02, 0.97]])

# define p(S|T,F)
cpdS = TabularCPD(variable='S', variable_card=2,
                  evidence=['T', 'F'], evidence_card=[2, 2],
                  values=[[0.99, 1.0, 0.92, 0.01],
                          [0.01, 0.0, 0.08, 0.99]])

# add probabilities to model
model.add_cpds(priorB, priorF, cpdG, cpdT, cpdS)
print('Model is valid: ', model.check_model())
print('True Model Edges: ',model.edges())
```

```
Model is valid:  True
True Model Edges:  [('B', 'G'), ('B', 'T'), ('F', 'G'), ('F', 'S'), ('T', 'S')]
```

Now that we have a defined model, let's generate some data using Gibbs sampling. Different amounts of samples are collected, in order to test the number of required samples for each of the later-implemented algorithms.

In [72]:
```
# generate sets of samples
sampler = GibbsSampling(model)
sample10 = sampler.sample(size=10, return_type='dataframe')
sample100 = sampler.sample(size=100, return_type='dataframe')
sample1000 = sampler.sample(size=1000, return_type='dataframe')
sample10000 = sampler.sample(size=10000, return_type='dataframe')
```

```
100%|██████████| 9/9 [00:00<00:00, 1132.47it/s]
100%|██████████| 99/99 [00:00<00:00, 1471.87it/s]
100%|██████████| 999/999 [00:00<00:00, 1577.62it/s]
100%|██████████| 9999/9999 [00:04<00:00, 2380.44it/s]
```

Parameter Learning

Still need to do these

Maximum Likelihood Estimation

Bayesian Estimation

Structure Learning

Now that we are able to fit conditional probabilities to a given bayesian-network structure, our next task is to generate a structure (Directed-Acyclic Graph) which best matches this data.

Network Scoring

The first family of methods we will explore which achieve this are *network scoring* techniques. These methods are similar to traditional numerical optimization, where we are trying to find the model structure M which maximizes the network score $p(M|D)$, which is the probability that our model fits the original data. This requires generating multiple possible models, and comparing their individual scores. In order to evaluate the score of a given model, we can compute an approximation of $p(M|D$ given that

$$p(M|D) \propto p(D|M)p(M).$$

From here we have to fit each model to the data, using one of the parameter learning methods described above. We will specifically use the bayesian parameter estimation method, since maximum likelihood estimation will tend towards more complex structures.

Exhaustive Search

The simplest method to search through model space is an exhaustive search, i.e. testing every possible combination of edges and directions. While this is simple to implement and guarantees finding the structure which absolutely the best score, this method can quickly become computationally intractable. Given that a network with n nodes has $\frac{n(n-1)}{2}$ distinct pairs of nodes, then for every pair there are three types of connections, meaning that there are a total of $3^{\frac{n(n-1)}{2}}$ possible connected graphs. This means that this approach is infeasable for all but the simplest of toy problems.

```
In [80]: es = ExhaustiveSearch(sample10000, scoring_method=K2Score(sample10000))
         start = time.time()
         best_model = es.estimate()
         end = time.time()
         print(best_model.edges())
         print("Elapsed Time:", end-start, ' sec')
```

```
[('B', 'T'), ('F', 'G'), ('F', 'S'), ('T', 'S')]
Elapsed Time: 180.61312747001648  sec
```

While simple, the exhaustive search is very slow. This method (along with other search methods) is also heavily tied to the samples used. As can be seen above, even for this simple network and 10,000 samples, the algorithm is only able to find four out of the five connections, and one of them has the incorrect independence relationship.

Hill Climbing

Another, more practical search method is a *Hill Climbing* search. In this method, we can start from a network with no connections and iteratively generate single connections which maximize the network score. This is possible because the overall score is comprised of additive terms of each node family. Due to this relationship, the change in score due to a single edge is not influenced by any other edge and we can use local heuristics entirely. Since we are only using local heuristics, we stop searching when we reach a local maximum.

Compare over # samples

```
In [73]: hc = HillClimbSearch(sample10, scoring_method=K2Score(sample10))
         start = time.time()
         best_model = hc.estimate()
         end = time.time()
         print(best_model.edges())
         print("Elapsed Time:", end-start, ' sec')
```

```
[('B', 'S'), ('T', 'S')]
Elapsed Time: 0.3474843502044678  sec
```

```
In [74]: hc = HillClimbSearch(sample100, scoring_method=K2Score(sample100))
         start = time.time()
         best_model = hc.estimate()
         end = time.time()
         print(best_model.edges())
         print("Elapsed Time:", end-start, ' sec')
```

```
[('F', 'G'), ('S', 'F')]
Elapsed Time: 0.3451557159423828  sec
```

```
In [76]: hc = HillClimbSearch(sample10000, scoring_method=K2Score(sample10000))
         start = time.time()
         best_modelhck2 = hc.estimate()
         end = time.time()
         print(best_modelhck2.edges())
         print("Elapsed Time:", end-start, ' sec')
```

```
[('B', 'T'), ('F', 'G'), ('F', 'S'), ('T', 'S')]
Elapsed Time: 0.46822309494018555  sec
```

Compare across score method

```
In [77]: hc = HillClimbSearch(sample10000, scoring_method=BDeuScore(sample10000, equival
         ent_sample_size=10000))
         start = time.time()
         best_modelhcBD = hc.estimate()
         end = time.time()
         print(best_modelhcBD.edges())
         print("Elapsed Time:", end-start, ' sec')
```

```
[('B', 'F'), ('B', 'S'), ('B', 'G'), ('G', 'F'), ('G', 'S'), ('F', 'S'), ('T',
'S'), ('T', 'B'), ('T', 'F'), ('T', 'G')]
Elapsed Time: 0.8934545516967773  sec
```

PC Algorithm

In [78]:
```
PCest = ConstraintBasedEstimator(sample10)
start = time.time()
skel, seperating_sets = PCest.estimate_skeleton(significance_level=0.01)
pdag = PCest.skeleton_to_pdag(skel, seperating_sets)
PCmodel = PCest.pdag_to_dag(pdag)
end = time.time()

print("Elapsed Time: ", end-start, " sec")
print("Undirected edges: ", skel.edges())
print("PDAG edges:       ", pdag.edges())
print("DAG edges:        ", model.edges())
```

```
Elapsed Time:  0.315032958984375  sec
Undirected edges:  [('G', 'F')]
PDAG edges:        [('G', 'F'), ('F', 'G')]
DAG edges:         [('B', 'G'), ('B', 'T'), ('F', 'G'), ('F', 'S'), ('T', 'S')]
```

Overall Comparison

In [97]:
```
'''inferenceTrue = VariableElimination(model)
hck2 = BayesianModel(best_modelhck2.edges())
hck2.add_cpds(best_modelhck2.get_cpds())
inferenceHCK2 = VariableElimination(best_modelhck2)
inferenceHCBD = VariableElimination(best_modelhcBD.edges())
inferencePC = VariableElimination(PCmodel.edges())

print('p(f|s=0,t=0)')
print('True:')
print(inferenceTrue.query(['F'], evidence={'S': 0, 'T': 0}))
print('Score (K2):')
print(inferenceHCK2.query(['F'], evidence={'S': 0, 'T': 0}))
print('Score (BDeau):')
print(inferenceHCBD.query(['F'], evidence={'S': 0, 'T': 0}))
print('PC:')
print(inferencePC.query(['F'], evidence={'S': 0, 'T': 0}))

print('p(b|s=0,t=0)')
print(inference.query(['B'], evidence={'S': 0, 'T': 0}))
print('p(f|s=0,t=1)')
print(inference.query(['F'], evidence={'S': 0, 'T': 1}))
print('p(b|s=0,t=1)')
print(inference.query(['B'], evidence={'S': 0, 'T': 1}))'''
```

Out[97]:
```
"inferenceTrue = VariableElimination(model)\nhck2 = BayesianModel(best_modelhck
2.edges())\nhck2.add_cpds(best_modelhck2.get_cpds())\ninferenceHCK2 = VariableE
limination(best_modelhck2)\ninferenceHCBD = VariableElimination(best_modelhcBD.
edges())\ninferencePC = VariableElimination(PCmodel.edges())\n\nprint('p(f|s=0,
t=0)')\nprint('True:')\nprint(inferenceTrue.query(['F'], evidence={'S': 0, 'T':
0}))\nprint('Score (K2):')\nprint(inferenceHCK2.query(['F'], evidence={'S': 0,
'T': 0}))\nprint('Score (BDeau):')\nprint(inferenceHCBD.query(['F'], evidence=
{'S': 0, 'T': 0}))\nprint('PC:')\nprint(inferencePC.query(['F'], evidence={'S':
0, 'T': 0}))\n\nprint('p(b|s=0,t=0)')\nprint(inference.query(['B'], evidence={'
S': 0, 'T': 0}))\nprint('p(f|s=0,t=1)')\nprint(inference.query(['F'], evidence=
{'S': 0, 'T': 1}))\nprint('p(b|s=0,t=1)')\nprint(inference.query(['B'], evidenc
e={'S': 0, 'T': 1}))"
```

In [ ]: