Analysing the saleability of cars through Bayesian Networks

Simone Montali - 983806

May 2021

Abstract

Provided with a **dataset** consisting of data about several cars, this research first tries to infer a **Bayesian Network** that accurately describes the problem and its probabilistic background, then, using the generated network, provides an analysis on the **interest** generated by the car for future saleability with respect to different features of the car. The inference is first computed with **variable elimination**, then an approximation through **Rejection Sampling** and **Likelihood Weighting** is compared to the initial results.

Contents

1 Introduction

The car market has been, in the last century, subject to a constant and strong growth. Nowadays, all kinds of features and gadgets have been added to all the car segments, from small city vehicles to luxury ones. Sometimes, though, to get a better grip on what the people prefer, abstracting from buttons, touch screens and custom horn sounds, **a simpler vision is preferable**. This is the reason why the analysis of a rather simple dataset through probabilistic reasoning could actually produce really interesting insights on how cars are chosen by the public.

2 Dataset

This dataset (click here) was created in 1997 by two researchers of the Joef Stefan Institute in Ljubljana, Slovenija to demonstrate a qualitative multi-criteria decision analysis (MCDA) method for decision making, DEX. The provided data consists of the classification of different features in ≈ 1800 cars. The target attribute is the **car acceptability**, a measure of the interest in the car demonstrated by the public. It is obvious that this dataset is a strong simplification of what actually happens when buying a car, and the fact that it was artificially crafted by someone doesn't help. Nonetheless, **it perfectly fits** the needs of a project like this one. The features that are stated for each car are the following: the buying price, the maintenance price, the number of doors, the seats, the size of the lug boot, the safety. Obviously, the numeric attributes like the prices are **bucketized**. Therefore, the final attribute list is the following:

attribute	**values**
acceptability	*unacc, acc, good, vgood*
buying	*vhigh, high, med, low*
maint	*vhigh, high, med, low*
doors	*2,3,4,5more*
people	*2,4,more*
lug_boot	*small, med, big*
safety	*low, med, high*

The data is in CSV format, so a car in the dataset looks like this:

```
vhigh,vhigh,3,2,small,high,unacc
```

3 Bayesian Network

Bayesian Networks allow us to **economically encode a probability distribution** over a set of variables, stating the **conditional independencies** across the different variables, a rather interesting notion in the analysis of this problem. They are a sufficient and compact specification of the full joint distribution. No Bayesian Network of the problem was available, but luckily the Python library `pgmpy` has exactly what is needed for us to obtain the network and its *Conditional Probability Tables* from the dataset.

3.1 Structure learning

To learn the model structure, i.e. a Directed Acyclic Graph, there are **two different families of techniques** that can be chosen: the constraint-based approaches, which search for a graph structure satisfying the independence assumptions that we observe in the empirical distribution; and the score-based approaches, which define an objective function for different models, and then search for a high-scoring model. [1] The approach I chose is the latter, needing a **scoring function** $s_D : M \to \mathbb{R}$, which maps a model $x \in M$ to a real score, based on how well it fits to the given dataset. Then, a search strategy traverses the space of all possible models and selects the one with optimal score. Several scoring functions are available, such as *BDeu*, *K2* and *BIC*. The last one, *Bayesian Information Criterion*, is a good asymptotic approximation of the likelihood of the training data. After having defined a scoring function, we can choose different search strategies, usually *local searches*, i.e. iterative search algorithms that keep improving a solution until a local maxima is reached. The algorithm I chose is **Hill Climb Search**, a rather simple algorithm that is able to explore the search space (while still not being extraordinarily performant) and find a good solution. To learn the network structure through `pgmpy`, we'll first have to import the data in a Pandas DataFrame:

```
import pgmpy
import pandas as pd
cars_data = pd.read_csv('data/car.data', names=["Buying", "Maintenance",
"Doors", "People", "LugBoot", "Safety", "Acceptability"])
```

Then, we'll just use *pgmpy*'s `HillClimbSearch` with a `BicScore`:

```
from pgmpy.estimators import HillClimbSearch
from pgmpy.estimators import BicScore

hc = HillClimbSearch(cars_data, scoring_method=BicScore(cars_data))
best_model = hc.estimate()
```

3.1.1 Learned graph

After some instants, the search finds its best guess of model. We can therefore print its **edges** with `best_model.edges()` obtaining the following result:

```
[('Buying', 'Maintenance'), ('Safety', 'People'), ('Safety', 'LugBoot'),
('Acceptability', 'Safety'), ('Acceptability', 'People'), ('Acceptability',
'Buying'), ('Acceptability', 'Maintenance'), ('Acceptability', 'LugBoot')]
```

which, in a more graphically appealing way, is the following graph:

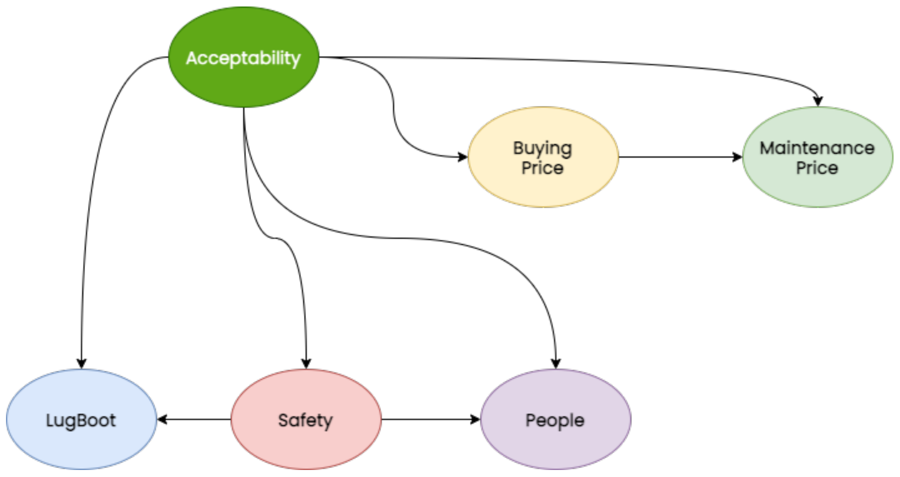

Figure 1: Bayesian network for the cars dataset

3.2 Parameter learning

Now that we've found a good structure for our graph, we'll have to find the values of the Conditional Probability Distributions. The library `pgmpy` is a great tool for this task too. If one would have to find a solution for this problem, it would be pretty natural to think about the **relative frequencies** of the variable states. This approach is, basically, what *Maximum Likelihood Estimation* does: as seen in section 17.1 of [1], the likelihood function for a given choice of parameters θ is the probability the model assigns the training data:

$$L(\boldsymbol{\theta} : \mathcal{D}) = \prod_{m} P(\xi[m] : \boldsymbol{\theta}) \tag{1}$$

In other words, we want to maximize the probability $P(data|model)$. This, in practice, is done by computing the state counts and dividing by the conditional sample size. MLE has a high possibility of overfitting on small very datasets. This, fortunately, is not our case, and we would otherwise have needed a different estimator, like Bayesian Parameter Estimation. We can estimate the parameters and print the CPDs with:

```
from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator

bayesian_model = BayesianModel(best_model.edges())
bayesian_model.fit(cars_data, estimator=MaximumLikelihoodEstimator)
for cpd in bayesian_model.get_cpds():
    print(cpd)
```

Having finally built our full joint distribution, we can proceed and **answer some questions through inference**.

4 Exact inference

4.1 Theoretical background

4.1.1 Summing out

At this moment, it is very well clear that the basic task for a probabilistic inference system would be, as the word says, performing **inference**, i.e. computing the posterior probability for a set of query variables, given some observed event in the form of evidence variable. Moreover, it is clear that by **summing out** the probabilities from the full joint distribution, we are able to compute any conditional probability. More specifically, a query $\mathbf{P}(X \mid \mathbf{e})$ can be always computed as:

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}) \tag{2}$$

4.1.2 Variable elimination

A crucial improvement in this algorithm is provided by **variable elimination**, which we can simplify as a form of **dynamic programming**, in which the factors get saved into an array to avoid recomputing the same thing multiple times.

4.2 Variable elimination with pgmpy

As always, `pgmpy` allows us to do so in a straightforward way:

```
from pgmpy.inference import VariableElimination
exact_inference = VariableElimination(bayesian_model)
```

Having instantiated the `exact_inference` object, we can then call queries on it. For example, a nice question to ask would be:

Is maintenance harder for 2-seaters than for family cars?

With just two lines of code, we can produce the probability tables for this query:

```
print(exact_inference.query(["Maintenance"],{'People':"2"}))
print(exact_inference.query(["Maintenance"],{'People':"4"}))
```

Maintenance level	**Probability**
very high	0.2975
high	0.2595
medium	0.2215
low	0.2215

(a) Maintenance on 2-seaters

Maintenance level	**Probability**
very high	0.2256
high	0.2450
medium	0.2646
low	0.2648

(b) Maintenance on 4-seaters

It looks like sport cars are not the best kind of choice if you want to stay cheap on oil and filters.

Are cars with high maintenance safe?

Safety	**Probability**
high	0.2793
medium	0.3240
low	0.3967

It looks like they are not!

5 Approximate inference

Exact inference for **singly connected graphs** (i.e. those in which there's at most one undirected path between any nodes) has a time and space complexity which is linear in the size of the network. This is good for graphs like the famous *Burglary Example*, but once we try inference on **multiply connected graphs**, we lose this nice property. Variable elimination can have **exponential time and space complexity** in the worst case, even when the number of parents per node is bounded. [2] So, what can we do?

5.1 Theoretical background

Approximate inference is the answer. It consists in **randomized sampling algorithms** (also known as *Monte Carlo*) that provide approximate answers having an accuracy depending on the number of samples generated. The obvious, first, step is **generating a sample** from a known probability distribution. This is basically how inference in real life is done: we run the experiment, count the results, compute the probabilities. It is also known as **stochastic approximation**, and for $\hat{N} \to N$ it will converge to the true probability. The standard deviation of the error in each probability will be proportional to $\frac{1}{\sqrt{n}}$.

5.1.1 Rejection sampling

Rejection sampling is probably the most straightforward way of proceeding. It can be used to compute conditional probabilities $P(X|e)$ by generating samples from the prior distribution, then **rejecting** those that do not match the evidence. Then, pretty easily, the estimate $\hat{P}(X = x|e)$ is obtained by counting how often $X = x$ occurs in the samples.

5.1.2 Likelihood weighting

Rejection sampling has a pretty notable downside: we are generating multiple samples that are inconsistent with our evidence, resulting in a non-trivial amount of wasted time. **Likelihood weighting** tries to solve that. The idea is simple: instead of throwing away the results that don't fit the evidence, we just weigh them less in the final count. The reason for doing that is simple: we would like to sample from the true posterior distribution, but usually this is too hard, as no polynomial-time approximation can exist. Now, a rather big question still has to be answered: *how do we compute the weights?* The solution is just the product of the conditional probabilities for the evidence variables given their parents:

$$w(\mathbf{z}) = \alpha \prod_{i=1}^{m} P\left(e_i \mid \text{ parents } \left(E_i\right)\right) \tag{3}$$

This approach can be **much more efficient** than rejection sampling, though it will lose its performance when n becomes higher: the noise that the low-weight samples add becomes more and more *distracting* as the number of samples increases. We know, according to Hoeffding's inequality, that

$$P(|s-p| > \epsilon) \leq 2e^{-2n\epsilon^2} \tag{4}$$

so we may be able to set an accuracy threshold for our approximate inference.

5.2 Approximate inference in pgmpy

As with any other interesting task, `pgmpy` allows us to perform these two. To keep things tidied up, a good idea would be creating a `SampleTester()` class having different methods for RS and LW. The processing method for rejection sampling would look like the following:

```python
def process_rs(self, size, evidence, query):
    rejection_sample = self.sampler.rejection_sample(evidence=evidence, size=size,
    return_type='recarray')[query]
    sample_probs = self.return_probs(rejection_sample)
    exact_result = self.exact_inference.query([query], dict(evidence)).values
    absolute_error = self.calculate_error(sample_probs, exact_result)
    return absolute_error
```

while the one for likelihood weighting would be as follows:

```python
def process_lws(self, size, evidence, query):
    likelihood_sample = self.sampler.likelihood_weighted_sample(evidence=evidence,
    size=size, return_type='recarray')
    sample_probs = self.return_weighted_probs(likelihood_sample[query],
    likelihood_sample['_weight'])
    exact_result = self.exact_inference.query([query], dict(evidence)).values
    absolute_error = self.calculate_error(sample_probs, exact_result)
    return absolute_error
```

Then, we just have to add some **utilities** for the error calculation and the sample counting:

```python
def return_probs(self, samples):
    # Get the unique values and their counts
    unique, counts = np.unique(samples, return_counts=True)
    # Divide the counts by the total number, getting a probability from 0 to 1
    counts = (counts/len(samples))
    # Zip the value and its probability in a dict
    return dict(zip(unique, counts))
def return_weighted_probs(self, samples, weights):
    unique = np.unique(samples)
    # Zero array for the weights sum, which we'll divide by the sum of weights
    counts = np.zeros(len(np.unique(samples)))
    iterator = np.nditer(samples, flags=['f_index'])
    for value in iterator:
        counts[value] += weights[iterator.index]
    counts = (counts/np.sum(weights))
    # Zip the value and its probability in a dict
    return dict(zip(unique, counts))
```

and that's it! We can test our methods with an example evidence $Safety = high, Maintenance = low$ and query the acceptability. The **results** are the following:

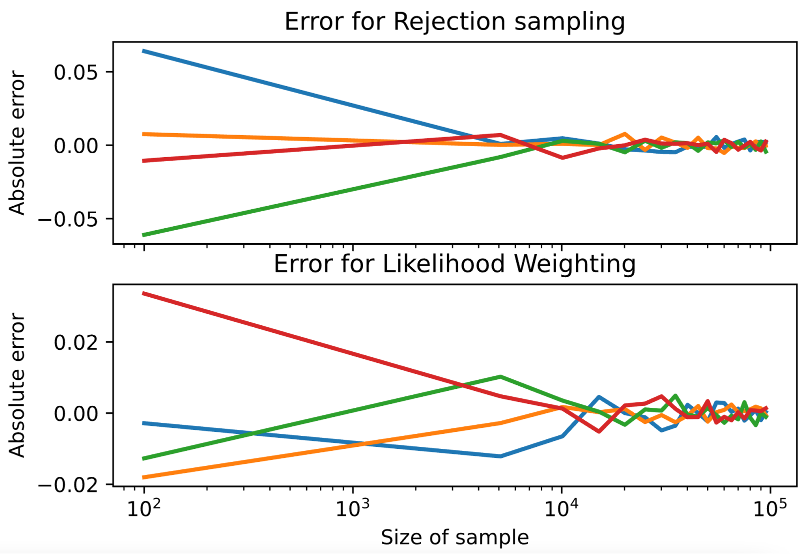

Figure 2: Errors achieved on each acceptability class for the two algorithms.

6 Conclusion

We have seen how Bayesian Networks can prove themselves as an **extraordinary tool** to summarize complex probabilistic domains. This could step up the process of decision making in multiple situations, while still being intrinsically elegant and simple. Tools like `pgmpy` make the whole job a breeze: the programmer just has to think about the problem, the computer will do the rest. That probably is how a truly *intelligent* system would perform. We have then seen how exact inference is - while still being computationally heavy - a great tool, and, moreover, how we can use **approximate inference** to achieve similar results with less, less work.

References

[1] Nir Friedman Daphne Koller. *Probabilistic Graphical Models: Principles and Techniques.* The MIT Press, 2009.

[2] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 edition, 2010.