PGM-TUTORIAL: EVALUATION OF THE PGMPY MODULE FOR PYTHON ON BAYES NETS

Mikael Eriksson, and John Folkesson,

Introduction

There are several libraries that implement probabilistic graphical models (PGMs) for many different languages. We will look at the pgmpy module for Python. Pgmpy was mainly developed by Ankur Ankan and Abinash Panda. They studied at the Indian Institute of Technology Varanasi and also co-authored the book ”Mastering Probabilistic Graphical Models using Python”.

This tutorial does not focus on a “state of the art” PGM but rather focuses on a basic understanding of how to build and evaluate the prediction performance of Bayesian PGMs as well as putting pgmpy to some tests using some real world student data.

Contents

This tutorial is based on two python files, the first being the code template pgm_tutorial.py and the other containing the data for the tutorial, pgm_tutorial_data.py.

The tutorial assignments are divided into 6 tasks that cover:

1. Fitting a naïve Bayes PGM to data
2. Queries / Inference
3. A model with reversed edges
4. Comparing model accuracy (versus distributions in data)
5. Structure search / optimization

Installation

Python 3 is used for this tutorial. The easiest way to install the dependencies is by using conda (see the file Environment_Setup). If you already have a python3 environment that you would like to continue using, you can paste the “pip” portion of the environment.yml file into a separate file and run:

pip install -r myFile.yml

Getting started

Activate your environment (as described in the Environment_Setup file), and run python interpreter from the shell, or use it to start whichever interpreter you prefer.

The data set provided for this tutorial

We gathered some actual data on a few cohorts of computer science engineering students here at KTH/CSC. We also performed some data cleaning like e.g. removing ”outliers” which although they had a registration on the computer science program, didn’t look a lot like the others in terms of what courses they had taken.

The features (variables) gathered on the data are:

- age: The age in years of the student when he/she started the program, quantized into one of '<=20' , '20-23' or '>23'
- gender: Binary gender, one of ’0’ or ’1’
- avg_cs: Average grade over the computer science courses, converted to a real number 0-5 and then quantized into one of '<2', '2<3', '3<4' or '4<5'
- avg_mat: Average grade over the math courses, same categorization as above
- delay: The time in years that the completion of the bachelor’s degree was delayed compared to nominal timing, categorized into one of '0', '1', '>=2' or 'NA'. Nominal zero delay means that if you start the bachelor’s thesis in the spring, you finish later the same year. ’>=2’ means that the bachelor’s degree was completed but with a delay of at least 2 years, while ’NA’ means that we don’t have any information on the completion of the degree.

One problem might be, for example, to construct a PGM that can predict the delay of the bachelor’s degree given information about the other variables.

Tasks

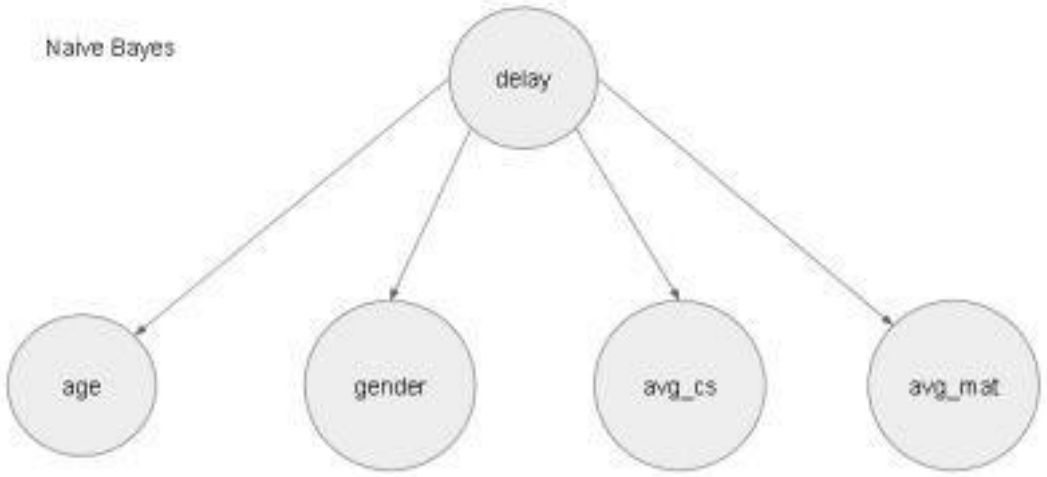
This tutorial is divided into 5 tasks. Each task has a block of python code that belongs to it, located in the file pgm_tutorial.py. The code is divided up into blocks on the form: ’’’ Task X -----------------

<Code>

’’’

By putting a # at the beginning of the first and the last line of a block, the code becomes active. There may also be a few interspersed theory questions in the tasks. You are expected to use the provided code as a starting point and then modify/add to it to answer the questions.

Task 1: Fitting a naive Bayes PGM to the data



Make the first task active and run it. It already shows how the PGM is set up. As can be seen one just specifies the list of edges in the graph. The fit method fits the data and when no estimator argument is given it uses the default MaximumLikelihoodEstimator.

The lines starting with

```
STATE_NAMES = model.cpds[0].state_names
```

prints out the variables and their list of values (states). It seems that pgmpy orders both the CPDs (based on the name of the owning variable) and the state names alphabetically. The indexes of the state names will be important later. The lib does under go development changing things like the order so the order should be checked.

The line:

```
print(model.cpds[0])
```

prints the model's first CPD (conditional probability distribution) or "tabular probability table" as they also call it. As mentioned, I believe everyone will see the CPD for age, as it is the one that comes first alphabetically.

Task 1.1. Warm up time: Looking at the printout of the CPD of the age variable answer the following: Which age group has the smallest representation among those that have delays longer than 2 years (">=2") on their bachelor's degree? (Why or why not is this information significant?)

Task 1.2. In the pgm_tutorial_data module I wrote a helper function that calculates distributions (relative frequencies) directly from data:

```
def ratio(data, posterior_lambda, prior_lambda)
```

The lambdas represent functions that filter for prior and posterior conditions. For example:

```
ratio(data, lambda t: t['avg_cs']=='4<5', lambda t:
t['avg_mat']=='4<5')
```

calculates the relative frequency of the average grade ”4<5” in computer science among those that had the same average in mathematics.

Verify whether the built in fit method set the marginal distributions of the delay variable (which has no parent nodes) to the relative frequencies in the data or not. Can you explain the result?

Task 1.3 (Optional). Verify in the same way as in task 1.2 for the CPD containing the variable avg_mat given delay, whether the probabilities are the same as the relative frequencies.

Task 2: Probability queries (inference)

The following is from the documentation for the query inference method (http://pgmpy.org/inference.html):

query(variables, evidence=None, elimination_order=None)

variables: list

list of variables for which you want to compute the probability

evidence: dict

a dict key, value pair as {var: state_of_var_observed} None if no evidence

elimination_order: list

order of variable eliminations (if nothing is provided) order is computed automatically

This is how you would query delay given a prior age (using default elimination ordering):

```
q = ve.query(variables = ['delay'], evidence = {'age': 0})
```

```
print(q)
```

If you are confused about the age value suddenly being an integer (and not a string), it happens that pgmpy queries work with discrete factors objects which are not embellished with information on state names and the like. Instead you use indexes into the state name lists. This is where you need to refer to the print out of STATE_NAMES mentioned above, to see what states the indexes correspond to. Remember that Python is zero-indexed.

If you want to see more than 4 decimals you can look at the factor's values like so:

```
q.values
```

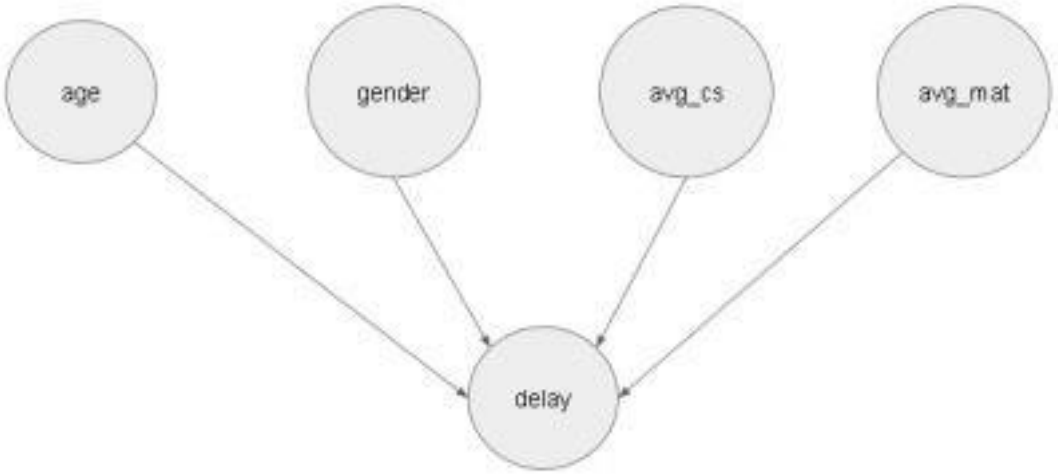
Task 2.1. Write a variable elimination query to infer the probability of zero delay for the age group '<=20'. What is the probability according to the PGM? How is this question different from the one in 1.1?

Task 2.2. Use the same kind of queries to determine which age group has the lowest probability of zero delays on the bachelor's degree and what is the probability? Which age group has the highest probability of zero delays on the bachelor's degree and what is the probability?

Task 2.3. Do the probabilities from task 2.2 come out the same when calculating relative frequencies directly on the data? Should they be the same with theoretical certainty when using this PGM structure and exact inference? Why or why not?

Task 2.4. MAP vs. MLE. Use the `map_query` (*variables=None* , *evidence=None*) method of VariableElimination to use another method for answering the question: which age group has the highest probability given zero delay. Do you get the same group as in 2.2? Why might the age group here have been different from 2.2?

Task 3: Inverting all the edges



In this PGM all the edges have been reversed so that delay directly depends on everything else. The intuition behind this might be that now we have really detailed information about how the variable of interest, delay, depends on everything else.

Task 3.1. Create the PGM pictured above and fit it to the data using the default ML-estimator (actually already done in the code)

Task 3.2. How many entries does the CPD for delay end up with in this PGM?

Task 3.3. How does the number of entries in the CPD for delay compare with the number of samples in the data set? How can the effect of this be seen in the probabilities in the CPD for delay?

Task 3.4. Examining the CPD for delay, what do the probability distributions look like for the cases where samples are missing? How many such rows are there? Hint: You could make a quadruple loop and then do some string matching – but there might be a nicer way. If you want it is actually possible to get a decent overview and feel of the rather big table, by replacing (str.replace) the vertical bar and cross characters with tabs, and then paste into Excel and optionally transposing it there. If this is too messy for you to handle, then it's OK to explain on theoretical grounds what the distributions should look like in these cases.

Task 3.5. Query the PGM to infer the marginal distribution for delay. Compare this to the distribution (relative frequencies) in the data. How well do they agree and what is the worst relative error among the probabilities? Should an exact inference always return the exact relative frequencies in data for this PGM structure? Why or why not?

Task 4: Comparing accuracy

The task at hand is to compare the accuracy of the two proposed PGM structures by means of taking the Kullback-Leibler divergence from the distributions in the data set (i.e. relative frequencies) to the resulting distributions of the PGMs, for a few queries. The first model is the naïve Bayes and the second is the one with edges reversed. The divergence function is implemented in scipy.stats.entropy. A second purpose is to create a set of performance data that can be further compared in the next task.

Look at the code given in task 4. It basically generates a number of queries for the distribution of delay given a randomly generated condition of evidence. It applies the queries to both of the explored models and compares how well they match the distribution in the data by means of the Kullback–Leibler divergence.

Task 4.1. Briefly recap the meaning of the Kullback-Leibler divergence, what it measures and why it isn't a "metric".

Task 4.2. In the original version of ppmpy we had "RuntimeWarning: invalid value encountered in true_divide" sometimes, but it didn't seem to prevent the analysis. However some of the divergence became 'inf' (infinity) for some cases. You may not get this now as pgmpy has changed the way it handles tables with no data. Can you see any of these cases and why the models could fail? Hint: recall something from a previous task. Observe that these cases are filtered out in divs2.

Task 4.3. Look at the lines:

```
n = 2
print([len([r for r in divs2 if len(r[0][1])==n]),
len([r for r in divs2 if len(r[0][1])==n and r[3] < r[5]]),
len([r for r in divs2 if len(r[0][1])==n and r[3] > r[5]]),
len([r for r in divs if len(r[0][1])==n and \
 not(math.isfinite(r[3]) and math.isfinite(r[5]))]),
sum(r[3] for r in divs2 if len(r[0][1])==n)])
```

They are counting and summing some things – what? How can the counts and sums be used to compare how well the two models perform? Hint: There are a few conditions here: a condition on the number of prior conditioned variables, a condition on which model had the smallest divergence

and a condition to check for infinite divergences. Check that you understand which condition is which.

Task 4.4. Increase the number of generated random queries from 20 to around 200 (that takes roughly half a minute on my laptop to process – you may try to make it faster by commenting out unnecessary prints). Use the lines from the previous task (modify them as needed) to fill in values for the following table:

N	M1 wins %	M2 wins %	Sum div M1	Sum div M2	Number of 'inf'
1					
2					
3					
4					

Where:

- N is the number of conditioned prior variables in the query
- "M1 wins %" is the ratio of queries (with N conditions), in which model 1 outperformed model 2 (in terms of having the smallest divergence from the distribution in the data)
- "M2 wins %" is the corresponding ration when model 2 outperformed model 1
- "Sum div M1" is the sum of divergences for model 1
- "Sum div M2" corresponding for model 2
- Number of 'inf' is the number of cases that got an infinite divergence

What trends do you see in the table when going from few prior conditions to many? Give some sort of intuition for the trends.

Task 4.5. Do the same as in 4.4 but using age as the target variable. Does the same pattern emerge as strongly?

Task 4.5. Create a list of performance data (like above) but with both delay and age as `targets(for target in ['delay', 'age']`), that together cover (at least) 200 randomly generated queries.

Task 5 : Finding a better structure

Task 5 will compare different BN models by means of the K2-Score. (See the paper, A Bayesian Method for the Induction of Probabilistic Networks from Data, Cooper and Herskovits, section 3.1.2 also Theorem 1 defines some of the notation. The proof is in the appendix. One is free to treat K2-Score as a black box that rates Bayes Nets.). It is not something central to our syllabus, but this gives you some insight into the topic of structure search.

Run the code for task 5.

Task 5.1. Which of the two models is the best according to the K2-score?

Task 5.2. What is the main idea with structure scoring methods and how are using them better than training on 100% of the data and doing no validation?

Task 5.3. Which model is the best according to the exhaustive search, when the gender variable is removed? Comment on the meaning of the structure – what conditional independencies is it suggesting?

Task 5.4. (Optional) When I run an exhaustive search on 4 variables it takes about 10 seconds. Make a combinatorial argument for how much time an exhaustive search over all 5 variables will take, based on how many graphs there will be to evaluate with 5 variables. You may ignore that loops are forbidden in Bayesian graphs. Hint: What is the total number of directed graphs with no multiple edges between two vertices, given n vertices?

Task 5.5. (Optional) Look at the example at http://pgmpy.org/estimators.html#hill-climb-search to help you use "hill climb search" on the data for 4 variables and for all 5 variables using the K2Score (don't worry about execution time here – this goes quickly). Do you get the same result as in 6.3? Comment on the resulting best PGM structure for all 5 variables. Do you get the same result when using BicScore? If you google the Bic score you may find claims that it tends to "underfit" the model when there is not enough data. How is this relevant to what happens here?

Task 5.6. Reflect over the use PGMs on the given dataset. Are using PGMs worth it in this case?