PGM_PyLib: A Python Library for Inference and Learning of Probabilistic Graphical Models

Jonathan Serrano-Pérez and L. Enrique Sucar

js.perez@inaoep.mx, esucar@inaoep.mx

Instituto Nacional de Astrofísica, Óptica y Electrónica
Coordinación de Ciencias Computacionales
Tonantzintla, Puebla

Contents

Chapter 1

Introduction

In this work the *Probabilistic Graphical Models Python Library* (**PGM_PyLib**) is described. It was written in Python for inference and learning of several classes of Probabilistic Graphical Models (PGM). The theory behind the different algorithms can be found in the book *Probabilistic Graphical Models Principles and Applications* [3].

1.1 Requirements

The library was implement to work correctly in Python 3[1]. Our test were run on Python 3.5.2, so, the library should work correctly in newer version of Python.

The library requires the Numpy[2] and SciPy[3] packages. Our test were run on Numpy 1.14.5 and SciPy 1.5.2, however, the library should work correctly in newer versions.

To verify the version of Python 3 that you have installed, you need to enter to the Python interpreter which immediately shows you the version you have installed, as shown in Figure 1.1. To verify that you have installed Numpy, first you need to type *import numpy as np*, if the python interpreter shows a error message then you have to install Numpy, else you have Numpy installed and you can check its version typing *np.version.version* as shown in Fig. 1.1. SciPy can be checked in the same way than Numpy.

1.2 Installation - Linux

First, you have to download the package in the following link https://github.com/jona2510/PGM_PyLib . After downloading the zip, you have to decompress it. Inside the zip you will find different examples which were written in Python, all them are shown along this work. Also you will find a folder called *PGM_PyLib* which is the library and contains the different implementations described in this work.

[1] https://www.python.org/download/releases/3.0/
[2] https://numpy.org/
[3] https://www.scipy.org/

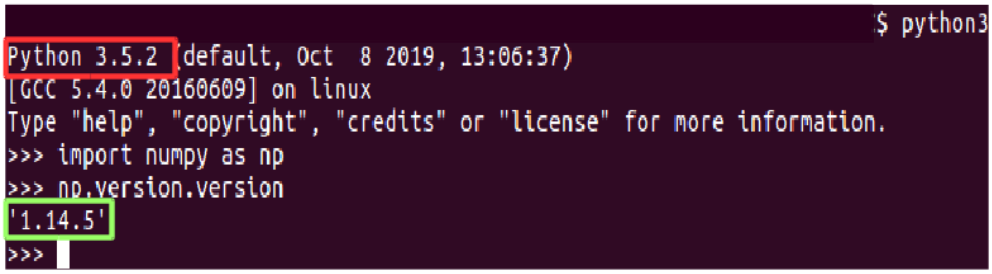

Figure 1.1: The version of Python is shown in the red box. The Numpy version is shown in the green box. Best seen in color.

The easiest way to use the library is to copy the full folder (*PGM_PyLib*) in your working directory, then you can use the different algorithms as shown along this work.

However, a better way is to add to the environment variable *PYTHONPATH* the directory where the library *PGM_PyLib* is. So you has to type in your terminal the following:

export PYTHONPATH="$PYTHONPATH:/the/full/path/folderX"

Note that the folder of *PGM_PyLib* is inside *folderX*. Furthermore, this only works in the current Linux terminal, and this is not permanent. However, you can add the previous line at the end of the file $\sim$ /*.bashrc* in order to be permanent.

Finally, try to run one of the examples, for example open a terminal in the path where the examples are, then type:

python3 exampleNBC.py

This example prints the percentage of correctly predicted instances (example of section 2.1.1)

Chapter 2

Bayesian Classifiers

2.1 Multiclass classification

2.1.1 Naive Bayes Classifier (NBC)

The Naive Bayes Classifier (NBC) is based in the assumption that all the attributes are independent given the class variable. So, each attribute A_i is conditionally independent of all other attributes given the class (C).

The classification problem can be formulated as:

$$Arg_C Max[log(P(C)) + log(P(A_1|C)) + ... + log(P(A_n|C))] \tag{2.1}$$

So, equation 2.3 was used to implement our variant of NBC, and the probabilities are estimated from data using maximum likelihood estimation.

2.1.1.1 *naiveBayes* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.naiveBayes.***naiveBayes** *(smooth=0.1, usePrior=True, meta="")*

Parameters:

- **smooth : float, default=0.1**: This value is used to smooth the estimations of all the probabilities in order to avoid probabilities of zero.
- **usePrior : bool, default=True**: It indicates whether to use the prior probabilities in the prediction phase.
- **meta : python dictionary, default=""**: if meta is equal to "" then the values that each attribute can take are obtained from the training set. Nevertheless, a dictionary with the values that each attribute takes can be provided, for example, *meta={0: ['a', 'b', 'c'], 1: ['1', '2']}*.

You can access to the following Attributes after training the classifier:

- **classes_ : ndarray of shape (n_classes,)**: It contains the different classes.

- **probsClasses : ndarray of shape (n_classes,)**: It contains the prior probabilities, that is, the probability of each class, $P(C)$.
- **valuesAtts : python dictionary**: Dictionary with the values that each attribute can take. The key of each item is the position of the attribute.
- **probsAtts : python dictionary**: Dictionary with the conditional probabilities of each attribute, that is, $P(A_i|C)$. The key of each item is the position of the attribute.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:
  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.
- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples)**: The predicted classes.
- **predict_log_proba***(testSet)* Return the scores obtained for each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The "log probability" of the instance for each class (that is, the estimation of equation 2.3). The classes are ordered as in **classes_**.
- **predict_proba***(testSet)* Return the probabilities of each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class, that is, it is equivalent to $exp(predict\_log\_proba)$ and normalized for each instance. The classes are ordered as in **classes_**.
- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:

- **real : ndarray of shape (n_samples)**: The true classes.
- **prediction : ndarray of shape (n_samples)**: The predicted classes.

The method returns:

- **float**: Percentage of correctly predicted classes.

2.1.1.2 Example of Naive Bayes Classifier

Below is an example of how to use the NBC. When this code is executed, it prints the percentage of correctly predicted instances using prior probabilities and the percentage without using the prior probabilities.

```
import numpy as np
import PGM_PyLib.naiveBayes as nb

np.random.seed(0)   # it is not necessary
# two classes
# 5 attributes

# 100 instances for training
data_train = np.random.randint(0,5,size=(100,5))
cl_train = np.random.randint(0,2,size=100)
# 50 instances for testing
data_test = np.random.randint(0,5,size=(50,5))
cl_test = np.random.randint(0,2,size=50)

# create the classifiers
c = nb.naiveBayes(smooth=0.1, usePrior=True)
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# ignore the Prior probabilities
c.usePrior = False
p = c.predict(data_test)
print(c.exactMatch(cl_test,p))
```

Listing 2.1: *exampleNBC.py*: Example of Naive Bayes Classifier

2.1.2 Sum-Naive Bayes Classifier

Sum-Naive Bayes Classifier (SumNBC) is similar to Naive Bayes Classifier (NBC) (section 2.1.1). Nevertheless, in this approach the prediction is the class with the highest score given by the sum of the probabilities of the different attributes.

Hence the classification problem is as follow:

$$Arg_C Max[P(C) + P(A_1|C) + P(A_2|C) + ... + P(A_n|C)] \tag{2.2}$$

So, equation 2.2 was used to implement SumNBC, and the probabilities are estimated from data using maximum likelihood estimation (see section 2.1.1).

2.1.2.1 *sumNaiveBayes* class

The class was implemented in python, so, the class with its default parameters is as follow:

*class PGM_PyLib.naiveBayes.***sumNaiveBayes** *(smooth=0.1, usePrior=True, meta="")*

Parameters:

- **smooth : float, default=0.1**: This value is used to smooth the estimations of all the probabilities in order to avoid probabilities of zero.
- **usePrior : bool, default=True**: It indicates whether to use the prior probabilities in the prediction phase.
- **meta : python dictionary, default=""**: if meta is equal to "" then the values that each attribute can take are obtained from the training set. Nevertheless, a dictionary with the values that each attribute takes can be provided, for example, *meta={0: ['a', 'b', 'c'], 1: ['1', '2']}*.

You can access to the following Attributes after training the classifier:

- **classes : ndarray of shape (n_classes,)**: It contains the different classes.
- **probsClasses : ndarray of shape (n_classes,)**: It contains the prior probabilities, that is, the probability of each class, $P(C)$.
- **valuesAtts : python dictionary**: Dictionary with the values that each attribute can take. The key of each item is the position of the attribute.
- **probsAtts : python dictionary**: Dictionary with the conditional probabilities of each attribute, that is, $P(A_i|C)$. The key of each item is the position of the attribute.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:
  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.
- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples)**: The predicted classes.

- **predict_proba***(testSet)* Return the probabilities of each class, where:

  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class. The classes are ordered as in **classes**.

- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:

  - **real : ndarray of shape (n_samples)**: The true classes.
  - **prediction : ndarray of shape (n_samples)**: The predicted classes.

  The method returns:

  - **float**: Percentage of correctly predicted classes.

2.1.2.2 Example of Sum-Naive Bayes Classifier

Below is an example of how to use the SumNBC. When this code is executed, it prints the percentage of correctly predicted instances using prior probabilities and the percentage without using the prior probabilities.

```
import numpy as np
import PGM_PyLib.naiveBayes as nb

np.random.seed(0)   # it is not necessary
# three classes
# 5 attributes

# 100 instances for training
data_train = np.random.randint(0,5,size=(100,5))
cl_train = np.random.randint(0,3,size=100)
# 50 instances for testing
data_test = np.random.randint(0,5,size=(50,5))
cl_test = np.random.randint(0,3,size=50)

# create the classifiers
c = nb.sumNaiveBayes(smooth=0.1, usePrior=True)
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# ignore the Prior probabilities
c.usePrior = False
p = c.predict(data_test)
```

```
print(c.exactMatch(cl_test,p))
```

Listing 2.2: *exampleSumNBC.py*: Example of Sum-Naive Bayes Classifier

2.1.3 Gaussian Naive Bayes Classifier (GNBC)

The Gaussian Naive Bayes Classifier (GNBC) is based in the assumption that all the attributes are independent given the class variable. So, each attribute A_i is conditionally independent of all other attributes given the class (C). Nevertheless, GNBC is able to handle continuous attributes while NBC (2.1.1 only nominal ones.

The classification problem can be formulated as:

$$Arg_C Max[log(P(C)) + log(P(A_1|C)) + ... + log(P(A_n|C))] \tag{2.3}$$

So, equation 2.3 was used to implement our variant of GNBC, and the probabilities are estimated from data using maximum likelihood estimation.

2.1.3.1 *GaussianNaiveBayes* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.naiveBayes.***GaussianNaiveBayes** *(smooth=0.1, usePrior=True, meta="")*

Parameters:

- **smooth : float, default=0.1**: This value is used to smooth the estimations of all the probabilities in order to avoid probabilities of zero.
- **usePrior : bool, default=True**: It indicates whether to use the prior probabilities in the prediction phase.
- **meta : python dictionary, default=""**: if meta is equal to "" then the values that each attribute can take are considered to be *numeric*. Nevertheless, a dictionary with the values that each attribute takes can be provided, for example, *meta={0: ['a', 'b', 'c'], 1: "numeric", 2: "numeric"}*, that indicates that there are one nominal attribute and two numeric.

You can access to the following Attributes after training the classifier:

- **classes_ : ndarray of shape (n_classes,)**: It contains the different classes.
- **probsClasses : ndarray of shape (n_classes,)**: It contains the prior probabilities, that is, the probability of each class, $P(C)$.
- **valuesAtts : python dictionary**: Dictionary with the values that each attribute can take. The key of each item is the position of the attribute.
- **probsAtts : python dictionary**: Dictionary with the conditional probabilities of each attribute, that is, $P(A_i|C)$, or a *ndarray* with the *mean* and *standard*

deviation if the attribute is numeric. The key of each item is the position of the attribute.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:
  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.
- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples)**: The predicted classes.
- **predict_log_proba***(testSet)* Return the scores obtained for each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, n_classes)**: The "log probability" of the instance for each class (that is, the estimation of equation 2.3). The classes are ordered as in **classes_**.
- **predict_proba***(testSet)* Return the probabilities of each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class, that is, it is equivalent to *exp*(*predict_log_proba*) and normalized for each instance. The classes are ordered as in **classes_**.
- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:
  - **real : ndarray of shape (n_samples)**: The true classes.
  - **prediction : ndarray of shape (n_samples)**: The predicted classes.

  The method returns:

  - **float**: Percentage of correctly predicted classes.

2.1.3.2 Example of Gaussian Naive Bayes Classifier

Below is a couple of examples of how to use the GNBC. When this code is executed, it prints the percentage of correctly predicted instances using prior probabilities and without using them, considering the nominal attributes. Later, a new classifier is trained which considers all attributes to be numeric.

```
import numpy as np
import PGM_PyLib.naiveBayes as nb

np.random.seed(0)   # it is not necessary
# two classes
# 5 attributes: 2 nominal and 3 numeric

# 200 instances for training
data_train = np.random.randint(0,5,size=(200,2))
data_train = np.concatenate([data_train, np.random.rand(200,3)],axis=1)
cl_train = np.random.randint(0,2,size=200)

# 100 instances for testing
data_test = np.random.randint(0,5,size=(100,2))
data_test = np.concatenate([data_test, np.random.rand(100,3)],axis=1)
cl_test = np.random.randint(0,2,size=100)

# create the dictionary with the values that each attribute can take
values = {0: [0,1,2,3,4], 1: [0,1,2,3,4], 2:"numeric", 3:"numeric", 4:"
    numeric"}


# Example 1, a GNB classifier is trained providing "values"
print("Results of GNBC:")
# create the classifiers
c = nb.GaussianNaiveBayes(smooth=0.1, usePrior=True, meta=values)
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# ignore the Prior probabilities
c.usePrior = False
p = c.predict(data_test)
print(c.exactMatch(cl_test,p))


# Example 2, a GNB classifier is trained considering all attributes to be
    numeric
print("Results of GNBC considering all attributes to be numeric:")
# create the classifiers
c2 = nb.GaussianNaiveBayes(smooth=0.1, usePrior=True, meta="")
# train the classifier
c2.fit(data_train, cl_train)
# predict
p = c2.predict(data_test)
# evaluation
```

```
print(c2.exactMatch(cl_test, p))

# ignore the Prior probabilities
c2.usePrior = False
p = c2.predict(data_test)
print(c2.exactMatch(cl_test,p))
```

Listing 2.3: *exampleGNBC.py*: Example of Gaussian Naive Bayes Classifier

2.1.4 Alternative Models: TAN, BAN

While NBC assumes that all attributes are independent given the class, there are models that incorporate some dependencies between the attributes. A couple of models are:

- **Tree augmented Bayesian Classifier (TAN)**.
- **Bayesian Network augmented Bayesian Classifier (BAN)**.

In this way, the classification problem can be formulated as:

$$Arg_C Max[log(P(C)) + log(P(A_1|Pa(A_1), C)) + ... + log(P(A_n|Pa(A_n), C))] \quad (2.4)$$

So, equation 2.4 was used to implement our variant of BAN, and the probabilities are estimated from data using maximum likelihood estimation.

2.1.4.1 *augmentedBC* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.augmented.***augmentedBC** *(algStructure="auto", smooth=0.1, usePrior=True)*

Parameters:

- **algStructure : ndarray of shape (n_features, n_features), default="auto"**: if algStructure is equal to "auto" then the structure is generated using the CLP-CMI algorithm (see section 5.1.2 for more details). However, a matrix which contains any directed acyclic graph can be provided, where there is *1* in the *i-th* row with *j-th* column if the *i-th* attribute is parent of the *j-th* attribute, *0* otherwise.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.
- **usePrior : bool, default=True**: It indicates whether to use the prior probabilities in the prediction phase.

You can access to the following Attributes after training the classifier:

- **classes_ : ndarray of shape(n_classes,)**: It contains the different classes.

- **probsClasses : ndarray of shape(n_classes,)**: It contains the prior probabilities, that is, the probability of each class, $P(C)$.

- **valuesAtts : python dictionary**: Dictionary with the values that each attribute can take. The key of each item is the position of the attribute.

- **probsAtts : python dictionary**: Dictionary with the conditional probabilities of each attribute, that is, $P(A_i|Pa(A_i), C)$. The conditional probabilities of each attribute are stored in a object of class **probsND**, see section 7.4.3. The key of each item is the position of the attribute.

- **structure : ndarray of shape (n_features+1, n_features+1)**: The structure (directed acyclic graph) that is used to estimate the probabilities of each attribute. Note, the last row and column correspond to the class variable.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:

  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.

- **predict***(testSet)* Return the prediction for each instance, where:

  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples)**: The predicted classes.

- **predict_log_proba***(testSet)* Return the scores obtained for each class, where:

  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, n_classes)**: The "log probability" of the instance for each class (that is, the estimation of equation 2.4). The classes are ordered as in **classes_**.

- **predict_proba***(testSet)* Return the probabilities of each class, where:

  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class, that is, it is equivalent to *exp*(*predict_log_proba*) and normalized for each instance. The classes are ordered as in **classes_**.

- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:

  - **real : ndarray of shape (n_samples)**: The true classes.
  - **prediction : ndarray of shape (n_samples)**: The predicted classes.

  The method returns:

  - **float**: Percentage of correctly predicted classes.

2.1.4.2 Example of Augmented Bayesian Classifier

Below is an example of how to use the Augmented Bayesian Classifier. When this code is executed, it prints the percentage of correctly predicted instances using prior probabilities and without using them.

```
import numpy as np
import PGM_PyLib.augmented as abc

np.random.seed(0)   # it is not necessary
# three classes
# 5 attributes

# 100 instances for training
data_train = np.random.randint(0,5,size=(100,5))
cl_train = np.random.randint(0,3,size=100)
# 50 instances for testing
data_test = np.random.randint(0,5,size=(50,5))
cl_test = np.random.randint(0,3,size=50)

# create the classifiers
c = abc.augmentedBC(algStructure="auto", smooth=0.1, usePrior=True)
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# ignore the Prior probabilities
c.usePrior = False
p = c.predict(data_test)
print(c.exactMatch(cl_test,p))
```

Listing 2.4: *exampleBAN.py*: Example of BAN

2.1.5 Semi-Naive Bayesian Classifiers

The idea of the Semi-Naive Bayesian Classifier is to eliminate or *join* attributes which are not independent given the class, in order to improve the performance of the classifier.

2.1.5.1 *semiNaive* class

The implementation of the class is based on the *Structural Improvement Algorithm* [3]. So, the class with its default parameters is as follow:

*class PGM_PyLib.semiNaive.***semiNaive** *(validation=0.8, epsilon=0.1, omega=0.1, nameAtts="auto", smooth=0.1, usePrior=True)*

Parameters:

- **validation : float, default=0.8**: This value is used to split the training set into two subset, in order to train a classifier with the first subset and evaluate it with the second, which is part of the algorithm.
- **epsilon : float, default=0.1**: The mutual information between an attribute and the class has to be at least **epsilon**, else the attribute is eliminated.
- **omega : float, default=0.1**: The conditional mutual information between two attributes given the class has to be lower than **omega**, else one attribute is eliminated or both attributes are combined.
- **nameAtts : ndarray of shape(n_features,), default="auto"**: It contains the name of attributes, this is useful to visualize the operations that were applied after training the classifier. If it is equal to "auto" the name of each attribute is its position.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.
- **usePrior : bool, default=True**: It indicates whether to use the prior probabilities in the prediction phase.

You can access to the following Attributes after training the classifier:

- **orderAtts : ndarray of shape (x,)**: It contains the name of the attributes after the modifications.
- **valuesAtts : python dictionary**: It contains the values that each attribute can take after training the classifier. The *key* of each item is the name of the attribute.
- **lvaluesAtts : python dictionary**: It contains the values that each attribute can take after training the classifier. The *key* of each item is the position of the attribute given by **orderAtts**.
- **opeNameAtts : ndarray of shape (x,)**: It contains the operations that where applied during the training.
- **NBC : naiveBayes object**: Semi-Naive Bayesian classifier uses a Naive Bayes Classifier, so, you can access to its attributes.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:

  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.

- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples)**: The predicted classes.

- **predict_log_proba***(testSet)* Return the scores obtained for each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The "log probability" of the instance for each class (that is, the estimation of equation 2.4). The classes are ordered as in **NBC.classes_**.

- **predict_proba***(testSet)* Return the probabilities of each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class, that is, it is equivalent to *exp*(*predict_log_proba*) and normalized for each instance. The classes are ordered as in **NBC.classes_**.

- **applyOperations***(data)* This method can be used after training the classifier. It applies the operations (**opeNameAtts**) to the data provided and return it as output. Where:
  - **data : ndarray of shape (n_samples, n_attributes)**: The data to be transformed.

  The method returns:
  - **ndarray of shape (n_samples, x)**: The data transformed.

- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:
  - **real : ndarray of shape (n_samples)**: The true classes.
  - **prediction : ndarray of shape (n_samples)**: The predicted classes.

The method returns:

- **float**: Percentage of correctly predicted classes.

2.1.5.2 Example of Semi-Naive Bayesian Classifier

Below is an example of how to use the Semi-Naive Bayesian Classifier. When this code is executed, it prints the percentage of correctly predicted instances using prior probabilities and without using them, also prints the operations that were applied.

```
import numpy as np
import PGM_PyLib.semiNaive as sn

np.random.seed(0)   # it is not necessary
# three classes
# 5 attributes

# 100 instances for training
data_train = np.random.randint(0,5,size=(100,5)).astype(str)
cl_train = np.random.randint(0,3,size=100)
# 50 instances for testing
data_test = np.random.randint(0,5,size=(50,5)).astype(str)
cl_test = np.random.randint(0,3,size=50)

# create the classifiers
c = sn.semiNaive(validation=0.8, epsilon=0.01, omega=0.01, smooth=0.1,
    nameAtts="auto", usePrior=True)
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# ignore the Prior probabilities
c.NBC.usePrior = False
p = c.predict(data_test)
print(c.exactMatch(cl_test,p))

# show the operations that were applied
print(c.opeNameAtts)
```

Listing 2.5: *exampleSNBC.py*: Example of Semi-Naive Bayes Classifier

2.1.5.3 Semi-Naive Bayes as feature selection

Taking advantage of our implementation, the *semiNaive* class can be used as feature selection. That is, after *training* the *classifier*, you can transform your data sets using the method **applyOperations**. In this way, you can use other classifier. Note, if you do this, you should consider providing to the classifier the different values that the attributes can take, that is, **lvaluesAtts**.

2.2 Multidimensional Classifiers

2.2.1 Bayesian Chain Classifiers

Bayesian Chain Classifiers (BCC) are a type of chained classifiers under a probabilistic framework, where the dependency relationships between variables are considered and represented as a directed acyclic graph.

In this way, the classification problem of the BCC is as follow:

$$\begin{array}{c} ArgMax_{C_1} P(C_1|\mathbf{Nei}(C_1), \mathbf{X}) \\ ArgMax_{C_2} P(C_2|\mathbf{Nei}(C_2), \mathbf{X}) \\ \dots\dots\dots\dots\dots\dots \\ ArgMax_{C_d} P(C_d|\mathbf{Nei}(C_d), \mathbf{X}) \end{array} \tag{2.5}$$

Where d is the number of multiclass problems that conform the multidimensional problem, $\mathbf{X}$ is the set of attributes and $\mathbf{Nei}(C_i)$ are the predictions of the *neighbours* of C_i which are added as attributes.

The neighbours that are considered are three:

- **Parents**: $Pa(C_i)$, which correspond to the original version of BCC.
- **Ancestors**: $Anc(C_i)$ which influence the predictions of C_i with its ancestors.
- **Children**: $Ch(C_i)$ which influence the predictions of C_i with its children.

2.2.1.1 *BCC* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.BCC.***BCC** *(chainType="parents", baseClassifier=naiveBayes(), structure="auto")*

Parameters:

- **chainType : {'parents', 'ancestors', 'children'}, default='parents'**: It indicates the neighbours that will influence the classifiers.
- **baseClassifier : classifier object, default=naiveBayes()**: Instance of a classifier which has to have at least the methods *fit()* and *predict()*. The default base classifier is a Naive Bayes Classifier with its default parameters, see section 2.1.1.
- **structure : ndarray of shape (d_classVariables, d_classVariables), default="auto"**: if it is equal to "auto" then the structure is generated using the CPL algorithm (see section 5.2.1 for more details) over the class variables. However, a matrix which contains any directed acyclic graph can be provided, where there is *1* in the *i-th* row with *j-th* column if the *i-th* class variable is parent of the *j-th* class variable, *0* otherwise.

You can access to the following Attributes after training the classifier:

- **structure : ndarray of shape (d_classVariables, d_classVariables)**: the structure which was used to chain the classifiers.

Methods of the class:

- **fit***(trainSet, cl)* this method trains the classifier, where:
  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.
  - **cl : ndarray of shape (n_samples, d_classVariables)**: The classes to which the instances are associated.
- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **ndarray of shape (n_samples, d_classVariables)**: The predicted classes.
- **predict_log_proba***(testSet)* Return the scores obtained for each class (only if available in the provided classifier), where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **list of shape (d_classVariables,)**: In each position of the list there is a *ndarray of shape (n_samples, n_classes)*, which contains the "log probability" of the instance for each class. Check the method **getClasses_()** to obtain the order of the classes.
- **predict_proba***(testSet)* Return the probabilities of each class (only if available in the provided classifier), where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:

  - **list of shape (d_classVariables)**: In each position of the list there is a *ndarray of shape (n_samples, n_classes)*, which contains the probability of the instance for each class. Check the method **getClasses_()** to obtain the order of the classes.
- **getClasses_()**: Method useful to obtain the classes of the different class variables. Some classifiers store the classes in a variable called *classes_* so the method **getClasses_()** has to be called in order to obtain the classes.

  The method returns:

  - **python dictionary**: which contains the classes that each class variable can take. The key of the item is the position of the class variable.

- **exactMatch***(real, prediction)*, Return the percentage of instances correctly predicted, where:

  - **real : ndarray of shape (n_samples)**: The true classes.
  - **prediction : ndarray of shape (n_samples)**: The predicted classes.

  The method returns:

  - **float**: Percentage of correctly predicted classes. An instance is correctly classified only if all the predictions for the class variables are correct.

2.2.1.2 Example Bayesian Chain Classifiers

Below is an example of how to use the BCC. When this code is executed,it prints the percentage of correctly predicted instances and the structure automatically generated.

```
import numpy as np
import PGM_PyLib.BCC as bcc
import PGM_PyLib.naiveBayes as nb

np.random.seed(0)   # it is not necessary
# 5 variable classes
# three classes for each variable class
# 7 attributes

# 300 instances for training
data_train = np.random.randint(0,5,size=(300,7))
cl_train = np.random.randint(0,3,size=(300,5))
# 100 instances for testing
data_test = np.random.randint(0,5,size=(100,7))
cl_test = np.random.randint(0,3,size=(100,5))

# create the classifiers
c = bcc.BCC(chainType="parents", baseClassifier=nb.naiveBayes(),
    structure="auto")
# train the classifier
c.fit(data_train, cl_train)
# predict
p = c.predict(data_test)
# evaluation
print(c.exactMatch(cl_test, p))

# show the structure
print(c.structure)
```

Listing 2.6: *exampleBCC.py*: Example Bayesian Chain Classifiers

2.3 Hierarchical Classification

2.3.1 HC with Bayesian Networks and Chained Classifiers

Hierarchical Classification (HC) with Bayesian networks and chained classifiers (BNCC) is a method which combines two strategies in order to predict the labels to which an instance is associated, while complains the *hierarchical constraint*.

Three different versions were proposed, which are different by the type of chained classifiers used. Additionally, the variant with independent local classifiers was implemented. All them use a Bayesian network, so, they are:

- **HCP**: Chained classifiers of parents.
- **HCA**: Chained classifiers of ancestors.
- **HCC**: Chained classifiers of children.
- **HBA**: Independent Local classifiers.

2.3.1.1 *BNCC* functions

Important: These implementations do NOT follow the same structure than previous classifiers. The files that are required by these methods are described in section 7.6. Furthermore, they require the *sklearn*[1] and *junctiontree*[2] packages.

The function with its default parameters is as follow (for HCA, HCC and HBA are the same parameters):

*function PGM_PyLib.hierarchicalClassification.BNCC.***HCP** *(header_in, train_in, test_in, tscore="SP", baseClassifier=RandomForestClassifier())*

Parameters:

- **header_in : string**: *Header* file which contains the information about the data and hierarchy.
- **train_in : string**: File which contains the data for training.
- **test_in : string**: File that contains the data for testing.
- **tscore : {"SP", "GLB"}, default="SP"**: It corresponds to the measure for scoring paths, that is, *SP* Sum of Probabilities or *GLB* Gain-Loose Balance.
- **baseClassifier : classifier object, default=RandomForestClassifier()**: Instance of a classifier which has to have the methods *fit, predit* and *predict_proba*, and the classes have to be stored in the attribute called *classes_*.

The function returns:

- **ndarray of shape (n_test_instances, n_classes)**: The predictions for each instance.

[1] https://scikit-learn.org/stable/install.html
[2] https://github.com/jluttine/junction-tree

2.3.1.2 Example of Hierarchical Classification with BNCC

Below is an example of how to use BNCC. In this code are printed the predictions obtained by the different variants. The files *header, train, test* used in this example are provided with the library.

```
import PGM_PyLib.hierarchicalClassification.BNCC as bncc
from sklearn.ensemble import RandomForestClassifier as rfc

# predit with HCP
p1 = bncc.HCP("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
    arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
print(p1)

#predict with HCA
p2 = bncc.HCA("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
    arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
print(p2)

#predict with HCC
p3 = bncc.HCC("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
    arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
print(p3)

#predict with HBA
p4 = bncc.HBA("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
    arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
print(p4)
```

Listing 2.7: *exampleBNCC.py*: Example of hierarchical classification with Bayesian Networks and Chain Classifiers

Chapter 3

Hidden Markov Models

3.1 Hidden Markov Models

A way of thinking about a Hidden Markov Models (HMM) is that it is a double stochastic process, that is, a hidden stochastic process that we can not directly observe, and a second stochastic process that produces the sequence of observations given the first process.

3.1.1 Evaluation

Evaluation consist in determining the probability of an observation sequence O given a model λ, that is, $P(O|\lambda)$. For this purpose, the *forward algorithm* has been implemented.

3.1.2 State Estimation

Finding the most probable sequence of states for an observation sequence can be interpreted in two ways. The first is to obtain the most probable state at each time step t. And the second is to obtain the most probable sequence of states. So, for the first case the function *MPS* will return the most probable state at time t, while the *Viterbi algorithm* has been implemented to solve the second case.

3.1.3 Learning

The parameters of a HMM model can be estimated from data. The *Baum Welch algorithm* was implemented for this purpose. However, the Expectation-Maximization (EM) principle is the handler of the Baum Welch algorithm in order to update its parameters.

3.1.4 *HMM* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.HMM.***HMM** *(states=None, observations=None, prior=None, transition=None, observation=None)*

Parameters:

- **states : python list, default=None**: List which contains the set of states.
- **observations : python list, default=None**: List which contains the set of observations.
- **prior : ndarray of shape (n_states,), default=None**: Vector of prior probabilities
- **transition : ndarray of shape (n_states, n_states), default=None**: Matrix of transition probabilities. The (i,j) cell indicates the probability of transiting to j-th state from the i-th state.
- **observation : ndarray of shape (n_states, m_observations), default=None**: Matrix of observation probabilities. The (i,j) cell indicates the probability of observing the j-th observation being in the i-th state.

You can access to the previous parameters as Attributes, except *observations* which changes its name by *obs*. They are useful when the model is learned from data.

Methods of the class:

- **forward***(O)* Implementation of the forward algorithm. Returns the probability of a sequence in the model. Where:
  - **O : python list**: The observation sequence that will be evaluated.

  The method returns:
  - **float**: The probability of the sequence.
- **forward_t***(t, O)* Applies the forward algorithm from time 1 up to t, hence, it returns $\alpha_t(i)$ of each state. Where:
  - **t : int**: time where the forward algorithm will be stopped. t is in range [1, length of O].
  - **O : python list**: The observation sequence.

  The method returns:
  - **ndarray of shape (n_states,)**: $\alpha_t(i)$ for each state.
- **backward_t***(t, O)* Applies the backward algorithm from time T up to t, where T is the length of O, hence, it returns $\beta_t(i)$ of each state. Where:
  - **t : int**: time where the backward algorithm will be stopped. t is in range [1, T].
  - **O : python list**: The observation sequence.

  The method returns:

  - **ndarray of shape (n_states,)**: $\beta_t(i)$ for each state.

- **MPS***(t, O)* Return the Most Probable State at time t, where:
  - **t : int**: Desired time. t is in range [1, length of O].
  - **O : python list**: The observation sequence.

  The method returns:

  - **state of states**: Return the most probable state.

- **viterbi***(O)* Return the most probable sequence of states given the observation sequence. Where:
  - **O : python list**: The observation sequence.

  The method returns:

  - **python list**: List with the most probable sequence of states
  - **float**: Score associated to the list.

- **learn(data,tol=0.01,hs=3,max_iter=10,initialization="uniform",seed=0)**: Method for learning the model from data. Where:
  - **data : list of lists**: List which contains observation sequences (lists).
  - **tol : float, default=0.01**: If the difference between the values of actual and estimated model are less than *tol*, then the learning converged and finish it.
  - **hs : int, default=3**: Number of hidden states.
  - **max_iter : int, default=10**: Maximum number of iterations. Finish the learning when this value is reached, that is, the learning has not converged yet.
  - **initialization : {"uniform", "random"}, default="uniform"**: Indicates how the initial parameters are initialized, that is, *uniform* probabilities or *random* probabilities.
  - **seed : int or "auto"**: This parameter is used when *initialization="random"*. If "auto" then the current numpy seed is used, else the *int* value is used as the seed.

  The method generates the model.

3.1.5 Example Hidden Markov Models

Below is a example of how to use HMM. Once the model is initialized, an observation sequence is evaluated using the *forward* algorithm, then the most probable state at each time t and the most probable sequence of states for the same observation sequence are obtained.

```python
import numpy as np
import PGM_PyLib.HMM as hmm

states = ["M1", "M2"]
obs = ["H", "T"]
PI = np.array( [0.5, 0.5] ) #prior probabilities
A = np.array( [[0.5, 0.5], [0.5, 0.5]] ) #transition probabilities
B = np.array( [[0.8, 0.2], [0.2, 0.8]] ) #observation probabilities

# Inializating the model with all its parameters
h = hmm.HMM(states=states, observations=obs, prior=PI, transition=A,
    observation=B)
O = ["H","H","T","T"] # observation sequence

# evaluating an observation sequence
print("Score of: H,H,T,T")
print(h.forward(O))

# obtaining the most probable state at each time t
lmps = [h.MPS(i,O) for i in range(1, len(O)+1) ]
print("Most probable state at each time t:")
print(lmps)

# obtaining the most probable sequence of states
mpss,score = h.viterbi(O)
print("Most probable sequence of states:")
print(mpss)
```

Listing 3.1: *exampleHMM.py*: Example of HMM

3.1.6 Example of learning a Hidden Markov Model

Below is a example of how to learn a HMM from data. After learning the model its attributes are shown. However, once the model is learned, the functions showed in the previous example can be used in this model.

```python
import numpy as np
import PGM_PyLib.HMM as hmm

data=[["H", "H", "T", "H", "T", "H", "T", "H", "T", "T"],
  ["T", "H", "H", "T", "T"],
  ["T", "H", "H", "T", "T", "H", "T"],
  ["H", "T", "T", "T", "T", "H", "T", "T", "T"],
  ["T", "T", "T", "H", "T", "T"]]
h = hmm.HMM() # empty model

# learning the model from data
h.learn(data,tol=0.001,hs=2,max_iter=100,initialization="random",seed=0)

print("Set of states:")
print(h.states)
print("Set of observations:")
print(h.obs)
print("Prior probabilities")
print(h.prior)
print("Transition probabilities")
```

```
print(h.transition)
print("Observation probabilities")
print(h.observation)
```

Listing 3.2: *exampleLearningHMM.py*: Example of Learning a HMM

Chapter 4

Markov Random Fields

Markov random fields (MRF) are undirected graphical models where each variable can take different values and is influenced probabilistically by the values of its neighbours.

In a MRF the main problem is to find the configuration of maximum probability. In this work, we are considering the Gibbs equivalence, that is, instead of maximizing the joint probability, we are minimizing the energy function.

Before showing the implemented classes, first let us introduce a description for MRF, which are described in the following way:

- **Structure**: The structure of a MRF can be *irregular* or *regular*.
- **Variables**: It has to do with whether all variables share the *same set* of states or if each variable has its *own set* of states.
- **Parameters**: The kind of parameters that are provided for the model, that is, *joint probabilities*, *potentials* or *local functions*.
- **Observations**: If the MRF has associated an observation or not.

Considering the previous description, we can describe the two variants implemented in this work:

- Regular; same set of states; local functions; without observation.
- Regular; same set of states; local functions; with observation.

That is, both work for regular MRF (two-dimensions), the variables share the same states set, the parameters are local functions, but one has associated an observations while the other not.

4.1 Regular Markov Random Fields

In a regular Markov random field (RMRF) its variables are arranged as a regular grid. And there is a neighborhood of order i, which is useful to define the number of variables to which each variable can be linked.

4.1.1 *RMRF* class

The class with its default parameters is as follow:
*class PGM_PyLib.MRF.***RMRF** *(states, rmrf)*

Parameters:

- **states : python list**: List which contains the set of states (integers). Or range of values as a string, for example "0-5" indicates the states $[0, 1, 2, 3, 4]$.
- **rmrf : ndarray of shape (x,y)**: A matrix which contains the initial values of the RMRF.

You can access to the following Attributes:

- **range : boolean**: Indicate if a range was given. If range was given, you can access to the following parameters:
  - **smin : int**: initial value of the range of states.
  - **smax : int**: final value of the range of states.
- **nstates : int**: number of states.
- **states : python list**: list with the states or the range provided.

Methods of the class:

- **inference***(Uf=smoothing, maxIterations=10, Temp=1.0, tempReduction=1.0, optimal="MAP")*: this method create the structure, where:
  - **Uf : python function**: Function that returns the local energy of a cell at position (row,col). Uf receives exactly 3 parameters:
    - *rmrf*: the matrix that contains the values of the RMRF.
    - *row, col*: position of the cell.
  - **maxIterations : integer, default=10**: Maximum number of iterations of the method until convergence.
  - **Temp : float, default=1**: This value is used as *temperature* in the variant *Simulated Annealing* (Temp > 0). It is the probability in the variant *Metropolis* ($0 <$ Temp < 1).
  - **tempReduction : float, default=1**: In each iteration the *Temp* is reduced in this way: $Temp = Temp * tempReduction$. So, $0 < tempReduction <= 1$.
  - **optimal : {"MAP","MPM"}, default="MAP"**: Optimal configuration, Maximum A posteriory Probability ("MAP") or Maximum Posterior Marginal ("MPM").

  The method returns:

  - **ndarray of shape (x, y)**: The RMRF after converging or when the maximum iterations is reached.

4.1.2 *RMRFwO* class

This class is similar to the previous one, however, this one has an extra attribute, *observation*. That is, the RMRF has associated an observation.

So, the class with its default parameters is as follow:
*class PGM_PyLib.MRF.***RMRFwO** *(states, rmrf, observation)*

Parameters:

- **states : python list**: List which contains the set of states (integers). Or range of values as a string, for example "0-5" indicates the states $[0, 1, 2, 3, 4]$.
- **rmrf : ndarray of shape (x,y)**: A matrix which contains the initial values of the RMRF.
- **observation : object**: A generic object which contains observation information. This information is passed as argument to *Uf function* in the *inference* method.

You can access to the following Attributes:

- **range : boolean**: Indicate if a range was given. If range was given, you can access to the following parameters:
  - **smin : int**: initial value of the range of states.
  - **smax : int**: final value of the range of states.
- **nstates : int**: number of states.
- **states : python list**: list with the states or the range provided.

Methods of the class:

- **inference***(Uf=smoothing, maxIterations=10, Temp=1.0, tempReduction=1.0, optimal="MAP")*: this method create the structure, where:
  - **Uf : python function**: Function that returns the local energy of a cell at position (row,col). Uf receives exactly 4 parameters:
    * *rmrf*: the matrix that contains the values of the RMRF.
    * *observation*: the object that contains the observation.
    * *row, col*: position of the cell.
  - **maxIterations : integer, default=10**: Maximum number of iterations of the method until convergence.
  - **Temp : float, default=1**: This value is used as *temperature* in the variant *Simulated Annealing* (Temp > 0). It is the probability in the variant *Metropolis* ($0 <$ Temp < 1).
  - **tempReduction : float, default=1**: In each iteration the *Temp* is reduced in this way: $Temp = Temp * tempReduction$. So, $0 < tempReduction <= 1$.

- **optimal : {"MAP","MPM"}, default="MAP"**: Optimal configuration, Maximum A posteriory Probability ("MAP") or Maximum Posterior Marginal ("MPM").

The method returns:

- **ndarray of shape (x, y)**: The RMRF after converging or when the maximum iterations is reached.

4.1.3 The variants of the method *inference*

The inference method, of the previous two classes, supports 3 variants which are executed in the following situations:

- **Iterative Conditional Modes (ICM)**: *tempReduction = 1* and *Temp = 1*.
- **Metropolis**: *tempReduction = 1* and *Temp != 1*.
- **Simulated Annealing**: *tempReduction != 1* and *Temp != 1*.

4.1.4 Example RMRF

Below is an example of a RMRF of order 1, where similar values between neighbours are preferred. First, the initial values of the RMRF are shown, then 3 configurations of the inference method are executed and the final result of each one is shown.

```
import numpy as np
from MRF import RMRF as rmrf

np.random.seed(0) # no mandatory

s = [i for i in range(6)] #s = "0-6"
r = np.random.randint(0,6,size=(5,7)) # RMRF of size 5x7
print("initial RMRF\n",r)

#ICM with MAP
mr = rmrf(s,r)
print("\nICM, MAP:")
r = mr.inference(maxIterations=100, Temp=1.0, tempReduction=1.0, optimal=
    "MAP")
print(r)
#Metropolis with MAP
mr = rmrf(s,r)
print("\nMetropolis, MAP:")
r = mr.inference(maxIterations=100, Temp=0.01, tempReduction=1.0, optimal
    ="MAP")
print(r)
#simulated annealing with MPM
mr = rmrf(s,r)
print("\nSimulated annealing, MPM:")
r = mr.inference(maxIterations=100, Temp=0.9, tempReduction=0.8, optimal=
    "MPM")
print(r)
```

Listing 4.1: *exampleRMRF.py*: Example of RMRF

Now, an example of RMRF with observation is shown, in this case a trade-off between "similar values between neighbours" and "similar values between the RMRF and the observation" is required. First, the initial values of the RMRF and observation are shown, then 3 configurations of the inference method are executed and the final result of each one is shown.

```
import numpy as np
from MRF import RMRFwO as mrf

s = [0,1]
r = np.zeros((4,4),dtype=int)
print("Initial RMRF\n",r)
obs=np.array([[0,0,0,0],[0,1,1,0],[0,1,0,0],[0,0,1,0]])
print("\nObservation\n",obs)


#ICM with MPM
mr = mrf(s,r,obs)
print("\nICM, MPM:")
r = mr.inference(maxIterations=100, Temp=1.0, tempReduction=1.0, optimal=
    "MPM")
print(r)
#Metropolis with MPM
mr = mrf(s,r,obs)
print("\nMetropolis, MPM:")
r = mr.inference(maxIterations=100, Temp=0.01, tempReduction=1.0, optimal
    ="MPM")
print(r)
#simulated annealing with MAP
mr = mrf(s,r,obs)
print("\nSimulated annealing, MAP:")
r = mr.inference(maxIterations=100, Temp=0.9, tempReduction=0.8, optimal=
    "MAP")
print(r)
```

Listing 4.2: *exampleRMRFwO.py*: Example of RMRF with observation

Chapter 5

Bayesian Networks

5.1 Learning Trees

5.1.1 Chow-Liu procedure (CLP)

The Chow-Liu procedure (CLP) [1,3] obtains the *skeleton* of a tree, but does not provide the directions of the arcs. That is, it estimates the Mutual Information (MI) (see section 7.1) between each pair of variables, and use the pairs of variables with the highest MI for building the tree. In this work, the directions of the arcs are given by selecting one variable as the root of the tree and assign directions to the arcs starting from this root.

5.1.1.1 *CLP_MI* class

The Chow-Liu prcedure was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.structures.trees.***CLP_MI** *(root=0, heuristic=False, smooth=0.1)*

Parameters:

- **root : int, default=0**: Position of the variable (given by the data table) which is selected as the root of the tree.
- **heuristic : bool, default=False**: If *True* then selects the node with more connections as the root of the tree, and the value of the attribute **root** is updated.
- **smooth : float, default=0.1**: This value is used as parameter for the estimation of the Mutual Information between the pairs of variables.

You can access to the following Attributes after creating the structure:

- **root : int**: The variable that is used as root of the tree.

Methods of the class:

- **createStructure***(data)*: this method create the structure, where:
  - **data : ndarray of shape (n_samples, n_variables)**: The data.

The method returns:

- **ndarray of shape (n_variables, n_variables)**: The matrix which contains a tree. The cell *(i,j)* is 1 if the *i-th* variable is parent of the *j-th* variable, 0 otherwise.

5.1.1.2 Example of CLP

Below is a example of how to use the Chow-Liu procedure. First, a tree is created where the variable 0 was selected as the root, then, a second structure was created, but, the heuristic was used in order to automatically select the root, finally, the root is shown.

```
import numpy as np
import PGM_PyLib.structures.trees as trees

np.random.seed (0)  # it is not necessary

# 7 variables
# 200 instances
data = np.random.randint(0,4,size=(200,3))
data = np.concatenate([data, np.random.randint(2,6,size=(200,4))],axis=1)

# create a instance of CLP_MI
clp = trees.CLP_MI(root=0, heuristic=False, smooth=0.1)
# create the structure
structure = clp.createStructure(data)
# show structure
print(structure)

# Use heuristic to automatically select the root of the tree
clp.heuristic = True
structure = clp.createStructure(data)
#show structure
print(structure)

#show the root node of the tree
print(clp.root)
```

Listing 5.1: *exampleCLP.py*: Example of CLP

5.1.2 CLP with Conditional Mutual Information

Chow-Liu procedure (CLP) with Conditional Mutual Information (CMI) follows the same idea than CLP (see section 5.2.1), nevertheless, in this case CMI is used instead of Mutual Information, that is, it is estimated the CMI between each pair of variables given an additional variable (this additional variable is the same for all the CMI estimations).

5.1.2.1 *CLP_CMI* class

The CLP_CMI was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.structures.trees.***CLP_CMI** *(root=0, heuristic=False, smooth=0.1)*

Parameters:

- **root : int, default=0**: Position of the variable (given by the data table) which is selected as the root of the tree.
- **heuristic : bool, default=False**: If *True* then selects the node with more connections as the root of the tree, and the value of the attribute **root** is updated.
- **smooth : float, default=0.1**: This value is used as parameter for the estimation of the Conditional Mutual Information.

You can access to the following Attributes after creating the structure:

- **root : int**: The variable that is used as root of the tree.

Methods of the class:

- **createStructure***(data, Z)*: this method create the structure, where:
  - **data : ndarray of shape (n_samples, n_variables)**: The data.
  - **Z : ndarray of shape (n_samples,)**: The additional variable which used for all the CMI estimations.

  The method returns:

  - **ndarray of shape (n_variables, n_variables)**: The matrix which contains a tree. The cell *(i,j)* is 1 if the *i-th* variable is parent of the *j-th* variable, 0 otherwise.

5.1.2.2 Example of CLP_CMI

Below is a example of how to use the CLP_CMI. First, a tree is created where the variable 0 was selected as the root, then, a second structure was created, but, the heuristic was used in order to automatically select the root, finally, the root is shown.

```
import numpy as np
import PGM_PyLib.structures.trees as trees

np.random.seed (0)  # it is not necessary

# 7 variables
# 200 instances
data = np.random.randint(0,4,size=(200,3))
data = np.concatenate([data, np.random.randint(2,6,size=(200,4))],axis=1)
#aditional variable
z = np.random.randint(1,5,size=(200))

# create a instance of CLP_CMI
clp_cmi = trees.CLP_CMI(root=0, heuristic=False, smooth=0.1)
# create the structure
structure = clp_cmi.createStructure(data, z)
```

```
# show structure
print(structure)

# Use heuristic to automatically select the root of the tree
clp_cmi.heuristic = True
structure = clp_cmi.createStructure(data, z)
#show structure
print(structure)

#show the root node of the tree
print(clp_cmi.root)
```

Listing 5.2: *exampleCLP-CMI.py*: Example of CLP

5.2 Learning DAGs

5.2.1 The PC algorithm

The PC algorithm can be seen as a two step method, first the *underlying undirected graph* is recovered, and second, the orientation of the edges are determined, in both steps, statistical test are required. Furthermore, the PC algorithm is able to recover DAG structures, nevertheless, it does not guarantee that all the edges have direction.

5.2.1.1 *PC* class

The PC algorithm was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.structures.DAG.***PC** *(n_adjacent=1, itest=chi2_ci_test(), itestDir=chi2_ci_test(), column_order="original", copy_data=True)*

Parameters:

- **n_adjacent : int, default=1**: The maximum number of variables adjacent to X (S) to evaluate $I(X, Y|S)$
- **itest : conditional_independence_test object, default=chi2_ci_test()**: Conditional independence test for obtaining the underlying undirected graph. The default object is a Pearson's chi-squared test for conditional independence with default parameters, see section 7.7.1. An external conditional independence test can be used, see section 8.2 for more details.
- **itestDir : conditional_independence_test object, default=chi2_ci_test()**: Conditional independence test for assigning direction to edges of the graph. The default object is a Pearson's chi-squared test for conditional independence with default parameters, see section 7.7.1. An external conditional independence test can be used, see section 8.2 for more details.
- **column_order : ndarray of shape (n_variables), default="original"**: The order in how the variables are iterated to apply the conditional independence tests.

Different orders can generated different graphs (structure and directions). If the value is "original", the variables are iterated in *original* order.

- **copy_data : bool, default True**: If True, then the matrix data will copy to a new variable, else, the operations are applied to the matrix data (columns of the matrix data are interchanged with respect to *column_order*).

You can access to the following Attributes after creating the structure:

- **structure : ndarray of shape (n_variables, n_variables)**: The matrix which contains the graph. The cell *(i,j)* is 1 if the *i-th* variable is parent of the *j-th* variable, 0 otherwise.

Methods of the class:

- **createStructure***(data)*: this method create the structure, where:
  - **data : ndarray of shape (n_samples, n_variables)**: The data.

  The method returns:
  - **ndarray of shape (n_variables, n_variables)**: The matrix which contains a tree. The cell *(i,j)* is 1 if the *i-th* variable is parent of the *j-th* variable, 0 otherwise.
- **orientationRules***()*: After creating the structure, this method can be called to assign direction to undirected edges of *structure* based on patterns [2]. First, patterns 3 and 4 are searched in the graph, then patterns 1 and 2.
- **orientationRules2***()*: After creating the structure, this method can be called to assign direction to undirected edges of *structure* based on patterns [2]. Patterns 1-4 are searched simultaneously.with a little preference to patterns 1 and 2 cause they only require 3 nodes.

5.2.1.2 Example of PC

Below is an example of how to use the PC algorithm. First, the DAG of Fig. 5.1 is used to generate instances (*data*), that is, we will try to recover that DAG from data. A conditional independence test *chi2_ci_test* (7.7.1) is used for learning the structure where maximum 3 conditional variables are considered, with a significance of 0.05, another *chi2_ci_test* is used to orient edges with a significance of 0.3. Then, we orient undirected edges based on patterns. Finally, the obtained graph is printed.

```
import numpy as np
from PGM_PyLib.structures.DAG import PC
from PGM_PyLib.stat_tests.ci_test import chi2_ci_test
from scipy.stats import bernoulli as ber

nv = 5000
np.random.seed(999999)  # it is not necessary

#the generation of variables x, z1, z2, y1, y2 is removed to reduce
```

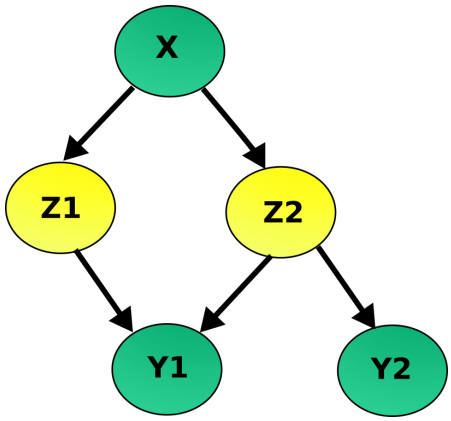

Figure 5.1: DAG from where the instance are generated

```
#   space, each one can take the values {0,1}.
#   The full example can be found in the library.
data = np.column_stack([x,z1,z2,y1,y2])

# conditional independence tests:
# tt: for learning the structure with a significance of 0.05:
# td: orient edges of the graph with a significance of 0.3:
tt = chi2_ci_test(significance=0.05, correction=False, lambda_=None,
    smooth=0.0)
td = chi2_ci_test(significance=0.3, correction=False, lambda_=None,
    smooth=0.0)

# Create an instance of PC
# for ci tests, maximum 3 conditional variables are considered
pct = PC(3, itest=tt, itestDir=td, column_order="original")
# generate structure with data
pct.createStructure(data)
# apply orientation rules for patterns
pct.orientationRules2()

# show the obtained graph
print(pct.structure)
```

Listing 5.3: *examplePC.py*: Example of PC

Chapter 6

Markov Decision Processes

6.1 Markov Decision Processes

Markov decision processes (MDP) model sequential decision problems, where a system evolves over time and is controlled by an agent. Solving a MDP, we get a *policy*, which indicates to an agent which action to select at each time step based on its current state.

6.1.1 *MDP* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.MDP.***MDP** *(reward, stateTransition, discountFactor=0.9)*

Parameters:

- **reward : ndarray or list of shape (n_states,)**: Each element of the list has to be a *python dictionary* where for each item, the key is the action that can be taken in the current state and the value is the reward. That is, a cell *(i, j)* contains the reward obtained if the agent is in *i-th* state and take the *j-th* action.

- **stateTransition : python list of shape (n_state)**: Each element of the list has to be a *python dictionary*, where for each item, the *key* is a "neighbour" state and the *value* is another *python dictionary* where for each item the key is an action and the value is the probability associated. That is, the cell *(i,j,k)* contains the probability of advancing to state j from state i after taking action k. Hence, $\sum_j(i,j,k) = 1, \forall i \in States, k \in Actions$.

- **discountFactor : float, default=0.9**: Discount factor for the value and policy iteration algorithms. $0 < discountFacor < 1$.

You can access to the previous parameters as Attributes, also to the following (after value or policy iteration algorithm was executed):

- **policy : ndarray of shape (n_states,)**: The policy for the agent, that is, each cell indicates the action that the agent must take.

Methods of the class:

- **valueIteration***(threshold, maxIter=-1)* Implementation of the value iteration algorithm. Returns the policy obtained. Where:
  - **threshold : float**: if the difference of the *values* on the current and previous iteration is lower than *threshold* then the process is terminated.
  - **maxIter : int, default=-1**: Maximum number of iteration allowed, if maxIter is lower than 0 then there is no limit.

  The method returns:
  - **policy : ndarray of shape (n_states,)**: The policy for the agent.
- **policyIteration***(maxIter=-1)* Implementation of the policy iteration algorithm. Returns the policy obtained. Where:
  - **maxIter : int, default=-1**: Maximum number of iteration allowed, if maxIter is lower than 0 then there is no limit.

  The method returns:
  - **policy : ndarray of shape (n_states,)**: The policy for the agent.

6.1.2 Example Markov Decision Process

Below is a example of how to use MDP. The grid depicted in Fig. 6.1 was used to model the MDP, *actions* {0:up, 1:down, 2:right, 3:left}, the *states* are numbered (0-10), for *rewards* advancing to:

- neighbour states has a reward of *-1*.
- forbidden state has a reward of *-100*.
- goal state has a reward of *+100*.

and, for the state transition function, a cell (i,j,k) has a high probability if it is expected that take the action k from state i will take us to j state, a lower one otherwise.

The MDP is initialized with the corresponding parameters, and both value policy iteration algorithms are applied and the obtained policies are shown.

```
import numpy as np
from PGM_PyLib.MDP import MDP

# Rewards:
# For state 0, we can take the actions down and right (1,2)
# For state 1, we can take the actions right and left (2,3)
# and so on.
R = np.array([
  {1:-1, 2:-1},      #0
  {2:-1, 3:-1},
  {1:-1, 2:100, 3:-1},  #2
  {1:-100, 3:-1},
```

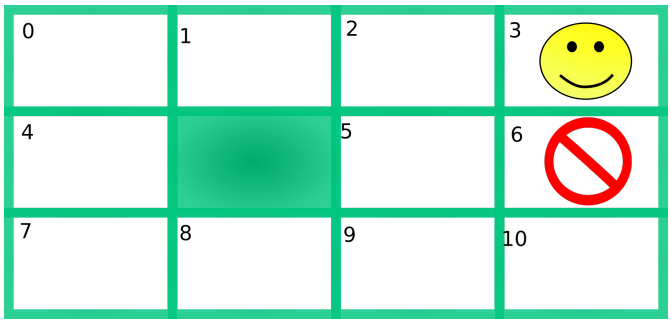

Figure 6.1: Grid world for example of MDP. The considered states are numbered.

```
  {0:-1, 1:-1},      #4
  {0:-1, 1:-1, 2:-100},
  {0:100, 1:-1, 3:-1},  #6
  {0:-1, 2:-1},
  {2:-1, 3:-1},      #8
  {0:-1, 2:-1, 3:-1},
  {0:-100, 3:-1},      #10
])

# state transition function (stf)
# stf has to be consistent with reward function, that is,
#   the actions in rewards has to be present in the stf for each state
# for example, for state 0, only the two actions are present (1,2)
# the same for state 1 (2,3) and so on.
FI = [
  { #   0-u, 1-d, 2-r, 3-l
    0: {1:0.1, 2:0.1},
    1: {1:0.1, 2:0.8},
    4: {1:0.8, 2:0.1}
  },
  {
    0: {2:0.1, 3:0.8},
    1: {2:0.1, 3:0.1},
    2: {2:0.8, 3:0.1}
  },
    #........
    # the state transition function is partially shown,
    #   the full example can be found in the library
]

# initialize the MDP
mdp = MDP( reward=R, stateTransition=FI, discountFactor=0.9 )

print("value iteration:")
policy = mdp.valueItetration(0.1)
print("policy:\n",policy)

print("\n policy iteration:")
policy = mdp.policyItetration()
print("policy:\n",policy)
```

Listing 6.1: *exampleMDP.py*: Example of MDP

Chapter 7

Annexes

7.1 Mutual Information

Equation 7.1 was used to estimate the Mutual Information (MI) between two varaibles.

$$MI(X;Y) = \sum_{y \in Y} \sum_{x \in X} P_{X,Y}(x,y) log \left(\frac{P_{X,Y}(x,y)}{P_X(x)P_Y(y)} \right) \tag{7.1}$$

where $P_{X,Y}$ is the joint probability of X and Y, and P_X and P_Y are the marginal probabilities of X and Y respectively.

7.1.1 *MI* function

The function that estimated the Mutual Information between two variables was implemented in Python. The function with its default parameters is as follow:

*function PGM_PyLib.utils.***MI** *(X, Y, smooth=0.1)*

Parameters:

- **X : ndarray of shape (n_samples,)**: The data of the first variable.
- **Y : ndarray of shape (n_samples,)**: The data of the second variable.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.

The function returns:

- **float**: Return the MI between two variables.

7.1.2 Example of Mutual Information

Below is an example of how to obtain the mutual information between a pair of variable.

```
import numpy as np
import PGM_PyLib.utils as utils

```

```
np.random.seed (0)  # it is not necessary

# 200 instances
X = np.random.randint(0,4,size=(200))
Y = np.random.randint(3,6,size=(200))

#estimate the MI between two varaibles
mi = utils.MI(X, Y, smooth=0.1)

#show the value obtained
print(mi)
```

Listing 7.1: Example Mutual Information

7.2 Conditional Mutual Information

Equation 7.2 was used to estimate the Conditional Mutual Information (CMI) between two variables given a third.

$$CMI(X;Y|Z) = \sum_{z \in Z} \sum_{y \in Y} \sum_{x \in X} P_{X,Y,Z}(x,y,z) log \left(\frac{P_Z(z) P_{X,Y,Z}(x,y,z)}{P_{X,Z}(x,z) P_{Y,Z}(y,z)} \right) \tag{7.2}$$

where $P_{X,Z}$, $P_{Y,Z}$ and $P_{X,Y,Z}$ are joint probabilities, and P_Z is the marginal probability of Z.

7.2.1 *CMI* function

The function that estimated the Conditional Mutual Information between two variables given a third was implemented in Python. The function with its default parameters is as follow:

*function PGM_PyLib.utils.***CMI** *(X, Y, Z, smooth=0.1)*

Parameters:

- **X : ndarray of shape (n_samples,)**: The data of the first variable.
- **Y : ndarray of shape (n_samples,)**: The data of the second variable.
- **Z : ndarray of shape (n_samples,)**: The conditional data.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.

The function returns:

- **float**: Return the CMI between two variables given a third.

7.2.2 Example of Conditional Mutual Information

Below is an example of how to obtain the conditional mutual information between a pair of variables given a third.

```python
import numpy as np
import PGM_PyLib.utils as utils

np.random.seed (0)  # it is not necessary

# 200 instances
X = np.random.randint(0,4,size=(200))
Y = np.random.randint(3,6,size=(200))
#conditional data
Z = np.random.randint(10,16,size=(200))

#estimate the CMI
cmi = utils.CMI(X, Y, Z, smooth=0.1)

#show the value obtained
print(cmi)
```

Listing 7.2: Example Conditional Mutual Information

7.3 Conditional Mutual Information Given a Conditional Set

This function is similar to CMI, nevertheless, in this case the conditional part is a set of variables.

Equation 7.3 was used to estimate the Conditional Mutual Information (CMI) between two variables given a **set of variables**.

$$CMI(X;Y|\mathbf{Z}) = \sum_{\mathbf{z}\in\mathbf{Z}}\sum_{y\in Y}\sum_{x\in X} P_{X,Y,\mathbf{Z}}(x,y,\mathbf{z}) log\left(\frac{P_{\mathbf{Z}}(\mathbf{z})P_{X,Y,\mathbf{Z}}(x,y,\mathbf{z})}{P_{X,\mathbf{Z}}(x,\mathbf{z})P_{Y,\mathbf{Z}}(y,\mathbf{z})}\right) \tag{7.3}$$

where $P_{X,\mathbf{Z}}$, $P_{Y,\mathbf{Z}}$ and $P_{X,Y,\mathbf{Z}}$ are joint probabilities, and $P_{\mathbf{Z}}$ is also a joint probability of $\mathbf{Z}$ set.

7.3.1 *CMI_setZ* function

The function that estimated the Conditional Mutual Information between two variables given a set of variables was implemented in Python. The function with its default parameters is as follow:

*function PGM_PyLib.utils.***CMI_setZ** *(X, Y, Z, smooth=0.1)*

Parameters:

- **X : ndarray of shape (n_samples,)**: The data of the first variable.
- **Y : ndarray of shape (n_samples,)**: The data of the second variable.
- **Z : ndarray of shape (n_samples, m_variables)**: The conditional data.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.

The function returns:

- **float**: Return the CMI between two variables given a set of variables.

7.3.2 Example of CMI_setZ

Below is an example of how to obtain the conditional mutual information between a pair of variables given a set of variables.

```
import numpy as np
import PGM_PyLib.utils as utils

np.random.seed (0)  # it is not necessary

# 200 instances
X = np.random.randint(0,4,size=(200))   # 4 values
Y = np.random.randint(3,6,size=(200))   # 3 values
#conditional data, 3 variables
Z = np.random.randint(10,16,size=(200,3))   #6 values each one

#estimate the CMI
cmi = utils.CMI_setZ(X, Y, Z, smooth=0.1)

#show the score obtained
print(cmi)
```

Listing 7.3: Example CMI_setZ

7.4 Estimation of probabilities of $P(A|C_1, C_2, ..., C_n)$

In order to estimate the conditional probabilities of a variable A given n variables, we create a special class to estimate the corresponding probabilities, $P(A|C_1, C_2, ..., C_n)$.

7.4.1 *probsND* class

The class was implemented in Python, so, the class with its default parameters is as follow:

*class PGM_PyLib.augmented.***probsND** *(variables, positions, smooth=0.1)*

Parameters:

- **variables : python dictionary**: Dictionary with the values that each variable can take.
- **positions : ndarray of shape (x,)**: each value in **positions** corresponds to a column/variable with respect to the matrix data (which is received as input in the method **estimateProbs**), in this way, the first value corresponds to A and the rest of values corresponde to $C_1, C_2, ..., C_n$. For example, the array $[0, 1, 2, 3, 4]$ indicates that will be estimated $P(V0|V1, V2, V3, V4)$, also arrays such as $[2, 0, 1, 3, 4]$ $[P(V2|V0, V1, V3, V4)]$ and $[3, 4]$ $[P(V3|V4)]$ are valid.
- **smooth : float, default=0.1**: This value is used to smooth the estimation of all the probabilities in order to avoid probabilities of zero.

You can access to the following Attributes after estimating the probabilities:

- **probabilities : ndarray**: Contains the conditional probabilities with respect to **positions**. It is a N-dimensional ndarray, where N is equal to the length of **positions**. The order of the dimensions is given by **positions**.

Methods of the class:

- **estimateProbs(*data*)**: Estimate the conditional probabilities and save the result in the attribute **probabilities**, where:
  - **data : ndarray of shape (n_samples, n_variables)**: the data used to estimate the conditional probabilities.

7.4.2 Example of estimation of conditional probabilities

Below is a code example. Inside the code, example 1 and 2 show how to estimate the conditional probabilities of two different configurations.

```
import numpy as np
import PGM_PyLib.augmented as pnd

np.random.seed (0)  # it is not necessary

# 5 variables
# variables 0 and 1 can take the values [7,8]
# variables 2,3,4 can take the values [10,11,12]
# 100 instances
data = np.random.randint(7,9,size=(100,3))
data = np.concatenate([data, np.random.randint(10,13,size=(100,2))], axis
    =1 )

# variables contains the values that each variable can take
variables = {0:[7,8], 1:[7,8], 2:[10,11,12], 3:[10,11,12], 4:[10,11,12]}

# Example 1: we want to estimate P(0|1,2,3,4)
positions = [0,1,2,3,4]
cpt = pnd.probsND(variables, positions, smooth=0.1)
cpt.estimateProbs(data)
#show the conditional probabilities
print(cpt.probabilities)

# Example 2: we want to estimate P(3|1,4)
positions = [3,1,4]
cpt2 = pnd.probsND(variables, positions, smooth=0.1)
cpt2.estimateProbs(data)
#show the conditional probabilities
print(cpt2.probabilities)

# The sum of aeA P(a|b,c,...) = 1, forall beB, forall ceC ...
#Example 3: below has to sum 1
print(cpt2.probabilities[0,0,0] + cpt2.probabilities[1,0,0] + cpt2.
    probabilities[2,0,0])

#Example 4: below has to sum 1
print(cpt2.probabilities[0,1,1] + cpt2.probabilities[1,1,1] + cpt2.
    probabilities[2,1,1])
```

Listing 7.4: *exampleCPT.py*: Example of estimation conditional probabilities

Table 7.1: Example of CPT generated by example 2. $P(V3|V1, V4)$

V1:		7			8		
V4:		10	11	12	10	11	12
	10	0.4426	0.2867	0.3987	0.3005	0.2287	0.1858
V3:	11	0.224	0.4965	0.4641	0.3498	0.3184	0.5398
	12	0.3333	0.2168	0.1373	0.3498	0.4529	0.2743

Table 7.1 is the conditional probability table (CPT) of $P(V3|V1, V4)$ which was estimated in example 2. We can see that the sum of $P(V0|V1 = 7, V4 = 10)$ is equal to 1 (Example 3), and that the sum of $P(V0|V1 = 8, V4 = 11)$ is equal to 1 (Example 4), this because one characteristic of the CPT is that satisfy equation 7.4.

$$\sum_{a \in A} P(a|c_1, c_2, ..., c_n) = 1, \forall c_1 \in C_1, \forall c_2 \in C_2, ..., \forall c_n \in C_n \tag{7.4}$$

7.4.3 Considerations for *augmentedBC* class

Previously was shown how the conditional probabilities are estimated. Hence, for the object that corresponds to the attribute A_i, you can access to its conditional probabilities using the attribute **probabilities** which follows the order given in **positions**. Note that the order of values in **positions** is as follow: first the attribute A_i, then its parents $Pa(A_i)$ and finally the class C. Also note that in the dictionary *variables* the item with the greatest *key* contains the values that the class can take.

7.5 Load Data

In this section, how to load data from ARFF and CSV files is shown. However, we suppose that data has already been preprocessed and there is no missing values. Furthermore, most of the algorithms require numeric or categorical features. Therefore, loading data is focused in only-numeric and only-categorical features. If you have a dataset with both categorical and numeric features, you first have to preprocess it in order to have only numeric or categorical features.

7.5.1 Load ARFF files

The package SciPy[1] is useful to load ARFF files. In Fig. 7.1 is depicted the content of the *exampleARFF.arff* file, which is the file from where the data will be loaded and a naive Bayes classifier will be trained. Bellow is the example.

```
from scipy.io import arff
import numpy as np
from PGM_PyLib.naiveBayes import naiveBayes as nbc

data, meta = arff.loadarff("exampleARFF.arff")   # load data
```

[1] https://www.scipy.org/

```
@RELATION exampleARFF

@ATTRIBUTE att1 {0,1}
@ATTRIBUTE att2 {0,1,2}
@ATTRIBUTE att3 {0,1}
@ATTRIBUTE class {true,false}

@DATA
0,1,0,true
1,1,1,false
0,2,0,true
1,0,0,false
0,2,1,false
1,0,1,true
```

Figure 7.1: Example of ARFF file (exampleARFF.arff).

```
att1,att2,att3,class
0,1,0,true
1,1,1,false
0,2,0,true
1,0,0,false
0,2,1,false
1,0,1,true
```

Figure 7.2: Example of CSV file (exampleCSV.csv).

```
data = np.array( data.tolist(), dtype=str ) # transform to numpy array
#variable class
cl = data[:,-1].copy()
# features
trainSet = data[:,:-1].copy().astype(int) # argument of astype could be
    changed by str or float, depending on the type of features
del data # delete data to save memory

# now, train for example a NBC
nb = nbc()  #default parameters
nb.fit(trainSet,cl)
# eval NBC in the training set
p = nb.predict(trainSet)
print(nb.exactMatch(cl,p))
```

Listing 7.5: Example of loading data from ARFF file.

7.5.2 Load CSV files

The package pandas[2] is useful to load CSV files. In Fig. 7.2 is depicted the content of the *exampleCSV.arff* file, which is the file from where the data will be loaded and a naive Bayes classifier will be trained. Bellow is the example.

```
import pandas as pd
from PGM_PyLib.naiveBayes import naiveBayes as nbc
```

[2]https://pandas.pydata.org

```
@RELATION my_relation

@ATTRIBUTE att1 numeric
@ATTRIBUTE att2 numeric
@ATTRIBUTE att3 numeric
@ATTRIBUTE att4 numeric
@ATTRIBUTE att5 numeric
@ATTRIBUTE att6 numeric
@ATTRIBUTE att7 numeric
@ATTRIBUTE att8 numeric
@ATTRIBUTE att9 numeric
@ATTRIBUTE class A/D,B/D,B/E
@ORDEN A,C,B,E,D

@DATA
```

Figure 7.3: Example of a header file for hierarchical classification.

```python

data = pd.read_csv("exampleCSV.csv")   # load data
data = data.to_numpy()  # transform to numpy array
# variable class
cl = data[:,-1].copy()
# features
trainSet = data[:,:-1].copy()
del data # delete data to save memory

# now, train for example a NBC
nb = nbc()  #default parameters
nb.fit(trainSet,cl)
# eval NBC in the training set
p = nb.predict(trainSet)
print(nb.exactMatch(cl,p))
```

Listing 7.6: Example of loading data from CSV file.

7.6 Files for hierarchical classification

Two different types of files are required for the implementations of hierarchical classification, the first is the *header* which follows the idea of the *arff* files, that is, first, the name of the relation, then the attributes, then a special attribute called *class* which contains the the relations of the hierarchy (A/D indicates that A is parent of D), and finally, *@ORDEN* which contains the order of the classes. An example is shown in Fig. 7.3.

The second file is which contains the data as described by the header file, notice that the classes are separated from the attributes by a semicolon. An example is shown in Fig. 7.4.

7.7 Statistical Test

In this section some implementations of statistical test are described.

```
0.3439,-3.6759,0.6856,0.1017,5.636,-0.657,1.0255,-1.1497,-0.4089;1,0,1,0,1
-0.4244,0.8794,-3.7009,-0.6478,1.8336,0.3729,0.7754,1.1846,-0.1223;1,0,1,0,1
```

Figure 7.4: Example of a data file for hierarchical classification.

7.7.1 Pearson's chi-squared test for Conditional Independence

First, let X and Y be two random variables, and $\mathbf{Z}$ be a set of random variables. So, $I(X, Y|\mathbf{Z})$ is a test which is *True* if X and Y are independent given $\mathbf{Z}$, *False* otherwise. Furthermore, a *contingency table* is a two-dimensional matrix that contains the observed frequencies in each combination of values from X and Y, it is recommended a value greater or equal than 5 in each cell.

This test wrappers the test *chi2_contingency*[3] from *scipy.stats*, that is, *chi2_contingency* is applied to each contingency table (X, Y) generated by the different combinations of values from $\mathbf{Z}$. Hence, there are as many individual test as combinations of $\mathbf{Z}$ values, however, if any individual test result is *False*, the result of the conditional test is *False*, else, the individual test statistics are summed and an overall χ^2 test is applied.

7.7.1.1 *chi2_ci_test* class

The class was implemented in python, so, the class with its default parameters is as follow (*correction* and *lambda_* are parameters for chi2_contingency, check its documentation for more details or let their values as default):

*class PGM_PyLib.stat_test.ci_test.***chi2_ci_test** *(significance=0.05, correction=False, lambda_=None, smooth=0.0)*

Parameters:

- **significance : float, default=0.05**: Significance for the different test. $0 < significance < 1$.
- **correction : bool, default=False**: *for chi2_contingency.* If True, and the degrees of freedom is 1, apply Yates correction for continuity.
- **lambda_ : float or str, default=None**: *for chi2_contingency.* By default, the statistic computed in this test is Pearson's chi-squared statistic. lambda_ allows a statistic from the Cressie-Read power divergence family to be used instead.
- **smooth : float, default=0.0**: This value is used to smooth the values of the contingency table.

You can access to the previous parameters as Attributes.

Methods of the class:

- **test***(X, Y, Z)* Applies the conditional test $I(X, Y|\mathbf{Z})$, where:

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.chi2_contingency.html

- **X : ndarray of shape (n_samples,)**: The data of variable X.
- **Y : ndarray of shape (n_samples,)**: The data of variable Y.
- **Z : ndarray of shape (n_samples, m_variables)**: The data of variables in **Z**.

The method returns:

- **bool**: *True* if X and Y are independent given **Z**, *False* otherwise.

7.7.1.2 Example

Below is an example of how to apply the Pearson's chi-squared test for conditional independence of two variables given a set of variables.

```
import numpy as np
from PGM_PyLib.stat_tests.ci_test import chi2_ci_test as cit

np.random.seed (0)  # it is not necessary

# 2000 samples
X = np.random.randint(0,2,size=(2000))  # 2 values
Y = np.random.randint(0,3,size=(2000))  # 3 values
#conditional data, 3 variables
Z = np.random.randint(0,2,size=(2000,3))    # 2 values each one

# initialize the ci test
t = cit(significance=0.05, correction=False,lambda_=None, smooth=0.0)

print("Are X and Y independent given the set Z?")
print( t.test(X,Y,Z) )
```

Listing 7.7: Example of conditional independence test.

Chapter 8

External Objects

The library is very flexible in some of its algorithms, that is, you can get external objects or develop your own objects for using them directly in the proper algorithm. Of course, the objects have to comply the requirements of the algorithm.

The templates of the objects are inside the folder *templates*, located at root of the package.

In the following sections the requirements for some objects are described. Furthermore, in the last section you will find information if you would like to submit your PGM algorithm to this library.

8.1 *Multiclass Classifier* Object

The multiclass classifiers requires two mandatory methods, *fit* which will train the classifier, and *predict* which will predict the class for the new instances. Furthermore, all the parameters for the classifier have to be provided in the constructor of the class.

So, the example class *my_classifier* with its default parameters is as follow:

class **my_classifier** *(my_parameter, my_other_parameter=0)*

This example requires two parameters, however, you can add as many parameters as your classifier requires for both training and predicting.

After training the classifier the following Attributes have to be available:

- **classes_ : ndarray of shape (n_classes,)**: It contains the different classes.
- **valuesAtts : python dictionary**: Dictionary with the values that each attribute can take. The key of each item is the position of the attribute.

Methods of the class, *fit* and *predict* are mandatory, the rest are optional:

- **fit***(trainSet, cl)* this method trains the classifier, where:
  - **trainSet : ndarray of shape (n_samples, n_features)**: The data for training.

  - **cl : ndarray of shape (n_samples)**: The classes to which the instances are associated.

- **predict***(testSet)* Return the prediction for each instance, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples)**: The predicted classes.

- (optional) **predict_log_proba***(testSet)* Return the scores obtained for each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The *log probability* of the instance for each class. The classes are ordered as in **classes_**.

- (optional) **predict_proba***(testSet)* Return the probabilities of each class, where:
  - **testSet : ndarray of shape (n_samples, n_features)**: The data to predict.

  The method returns:
  - **ndarray of shape (n_samples, n_classes)**: The probability of the instance for each class. The classes are ordered as in **classes_**.

Of course, you can add as many methods as you require. The template that you can use as guideline is *my_classifier.py*

8.2 *Conditional Independence Test* Object

The conditional independence test requires just one mandatory method, *test*, which will execute the test $I(X, Y|\mathbf{Z})$. All the parameters for the test have to be provided in the constructor of the class.

The example class *my_ci_test* with its default parameters is as follow:
class **my_ci_test** *(significance=0.05)*

This example requires one parameter, however, you can add as many parameters as your conditional independence test requires.

This object does not requires special attributes.

Methods of the class, *test* is mandatory:

- **test***(X, Y, Z)* Applies the conditional test $I(X, Y|\mathbf{Z})$, where:
  - **X : ndarray of shape (n_samples,)**: The data of variable X.
  - **Y : ndarray of shape (n_samples,)**: The data of variable Y.
  - **Z : ndarray of shape (n_samples, m_variables)**: The data of variables in **Z**.

  The method returns:

  - **bool**: *True* if X and Y are independent given **Z**, *False* otherwise.

Of course, you can add as many methods as you require. The template that you can use as guideline is *my_ci_test.py*

8.3 Submission of algorithms

Submission of new PGM algorithms are very welcome. However, they have to comply with the requirements of the specific *external object*. Furthermore, you have to provide us the following information in order to publish your algorithm:

1. **Code**: The code of the algorithm, it is highly recommended that it is commented.
2. **Example**: The code of how to use the algorithm (at least one example).
3. **Description**: A very brief description of the algorithm, the description of the class (parameters, attributes, methods) and description of the example. Preferably in latex.
4. **Requirements**: List of packages and/or software that the algorithm requires.
5. **References**: The reference(s) where the algorithm is reported (only in case that it is not reported in [3]). Bibtex format.
6. **Consent to publish**: PGM_PyLib is distributed under the *GNU public license v3.0*, hence, you have to agree that your algorithm also will be distributed under the same license.

Consider that comply with the previous requisites does not guarantee that your algorithm will be accepted and publish. However, we will inform you the decision made.

Feel free to contact us before making your submissions. We will try to reply as soon as possible.

Bibliography

[1] Chow, C., and Liu, C. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory 14*, 3 (May 1968), 462–467.

[2] Meek, C. Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence* (San Francisco, CA, USA, 1995), UAI'95, Morgan Kaufmann Publishers Inc., p. 403–410.

[3] Sucar, L. E. *Probabilistic Graphical Models Principles and Applications*, 1 ed. Springer-Verlag London, 2015.