



INAOE

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

©INAOE 2020
All rights reserved.



Contents

1	Introduction	2
1.1	Requirements	2
1.2	Installation - Linux	3
2	Bayesian Classifiers	4
2.1	Multiclass classification	4
2.1.1	Naive Bayes Classifier (NBC)	4
2.1.1.1	<i>naiveBayes</i> class	4
2.1.1.2	Example of Naive Bayes Classifier	6
2.1.2	Sum-Naive Bayes Classifier	6
2.1.2.1	<i>sumNaiveBayes</i> class	7
2.1.2.2	Example of Sum-Naive Bayes Classifier	8
2.1.3	Gaussian Naive Bayes Classifier (GNBC)	9
2.1.3.1	<i>GaussianNaiveBayes</i> class	9
2.1.3.2	Example of Gaussian Naive Bayes Classifier	11
2.1.4	Alternative Models: TAN, BAN	12
2.1.4.1	<i>augmentedBC</i> class	12
2.1.4.2	Example of Augmented Bayesian Classifier	14
2.1.5	Semi-Naive Bayesian Classifiers	14
2.1.5.1	<i>semiNaive</i> class	15
2.1.5.2	Example of Semi-Naive Bayesian Classifier	17
2.1.5.3	Semi-Naive Bayes as feature selection	17
2.2	Multidimensional Classifiers	18
2.2.1	Bayesian Chain Classifiers	18
2.2.1.1	<i>BCC</i> class	18
2.2.1.2	Example Bayesian Chain Classifiers	20
2.3	Hierarchical Classification	21
2.3.1	HC with Bayesian Networks and Chained Classifiers	21
2.3.1.1	<i>BNCC</i> functions	21
2.3.1.2	Example of Hierarchical Classification with BNCC	22
3	Hidden Markov Models	23
3.1	Hidden Markov Models	23
3.1.1	Evaluation	23
3.1.2	State Estimation	23

3.1.3	Learning	23
3.1.4	<i>HMM</i> class	23
3.1.5	Example Hidden Markov Models	25
3.1.6	Example of learning a Hidden Markov Model	26
4	Markov Random Fields	28
4.1	Regular Markov Random Fields	28
4.1.1	<i>RMRF</i> class	29
4.1.2	<i>RMRFwO</i> class	30
4.1.3	The variants of the method <i>inference</i>	31
4.1.4	Example RMRF	31
5	Bayesian Networks	33
5.1	Learning Trees	33
5.1.1	Chow-Liu procedure (CLP)	33
5.1.1.1	<i>CLP_MI</i> class	33
5.1.1.2	Example of CLP	34
5.1.2	CLP with Conditional Mutual Information	34
5.1.2.1	<i>CLP_CMI</i> class	34
5.1.2.2	Example of CLP_CMI	35
6	Annexes	37
6.1	Mutual Information	37
6.1.1	<i>MI</i> function	37
6.1.2	Example of Mutual Information	37
6.2	Conditional Mutual Information	38
6.2.1	<i>CMI</i> function	38
6.2.2	Example of Conditional Mutual Information	38
6.3	Estimation of probabilities of $P(A C_1, C_2, ..., C_n)$	39
6.3.1	<i>probsND</i> class	39
6.3.2	Example of estimation of conditional probabilities	40
6.3.3	Considerations for <i>augmentedBC</i> class	41
6.4	Files for hierarchical classification	41

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* [2].

1.1 Requirements

The library was implement to work correctly in Python 3¹. Our test were run on Python 3.5.2, so, the library should work correctly in newer version of Python.

The library requires the Numpy² package. Our test were run on Numpy 1.14.5, 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.

¹<https://www.python.org/download/releases/3.0/>

²<https://numpy.org/>

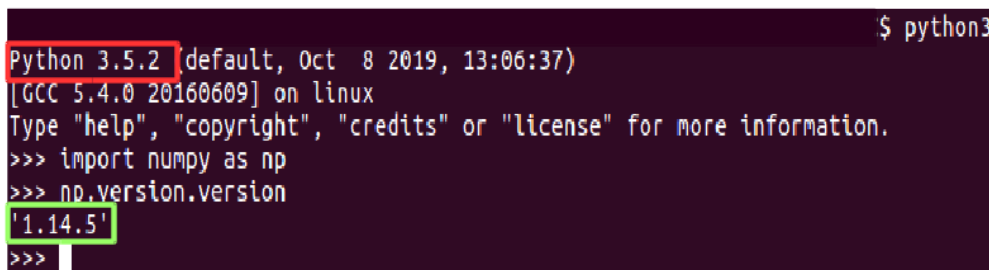
A terminal window with a dark background. The prompt is '\$ python3'. The first line of output is 'Python 3.5.2 [default, Oct 8 2019, 13:06:37]', which is enclosed in a red rectangular box. The next line is '[GCC 5.4.0 20160609] on linux'. The following line is 'Type "help", "copyright", "credits" or "license" for more information.'. Then, the user enters '>>> import numpy as np'. The next line is '>>> np.version.version', and the output is ''1.14.5'', which is enclosed in a green rectangular box. The prompt '>>>' is visible at the bottom.

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.

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.

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 `~/.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:

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

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 $\exp(\text{predict_log_proba})$ 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.

```

1 import numpy as np
2 import PGM_PyLib.naiveBayes as nb
3
4 np.random.seed(0)    # it is not necessary
5 # two classes
6 # 5 attributes
7
8 # 100 instances for training
9 data_train = np.random.randint(0,5,size=(100,5))
10 cl_train = np.random.randint(0,2,size=100)
11 # 50 instances for testing
12 data_test = np.random.randint(0,5,size=(50,5))
13 cl_test = np.random.randint(0,2,size=50)
14
15 # create the classifiers
16 c = nb.naiveBayes(smooth=0.1, usePrior=True)
17 # train the classifier
18 c.fit(data_train, cl_train)
19 # predict
20 p = c.predict(data_test)
21 # evaluation
22 print(c.exactMatch(cl_test, p))
23
24 # ignore the Prior probabilities
25 c.usePrior = False
26 p = c.predict(data_test)
27 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:

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

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.

```

1 import numpy as np
2 import PGM_PyLib.naiveBayes as nb
3
4 np.random.seed(0)    # it is not necessary
5 # three classes
6 # 5 attributes
7
8 # 100 instances for training
9 data_train = np.random.randint(0,5,size=(100,5))
10 cl_train = np.random.randint(0,3,size=100)
11 # 50 instances for testing
12 data_test = np.random.randint(0,5,size=(50,5))
13 cl_test = np.random.randint(0,3,size=50)
14
15 # create the classifiers
16 c = nb.sumNaiveBayes(smooth=0.1, usePrior=True)
17 # train the classifier
18 c.fit(data_train, cl_train)
19 # predict
20 p = c.predict(data_test)
21 # evaluation
22 print(c.exactMatch(cl_test, p))
23
24 # ignore the Prior probabilities
25 c.usePrior = False
26 p = c.predict(data_test)

```

```
27 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:

$$\text{Arg}_C \text{Max}[\log(P(C)) + \log(P(A_1|C)) + \dots + \log(P(A_n|C))] \quad (2.3)$$

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 $\exp(\text{predict_log_proba})$ 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.

```

1 import numpy as np
2 import PGM_PyLib.naiveBayes as nb
3
4 np.random.seed(0)    # it is not necessary
5 # two classes
6 # 5 attributes: 2 nominal and 3 numeric
7
8 # 200 instances for training
9 data_train = np.random.randint(0,5,size=(200,2))
10 data_train = np.concatenate([data_train, np.random.rand(200,3)],axis=1)
11 cl_train = np.random.randint(0,2,size=200)
12
13 # 100 instances for testing
14 data_test = np.random.randint(0,5,size=(100,2))
15 data_test = np.concatenate([data_test, np.random.rand(100,3)],axis=1)
16 cl_test = np.random.randint(0,2,size=100)
17
18 # create the dictionary with the values that each attribute can take
19 values = {0: [0,1,2,3,4], 1: [0,1,2,3,4], 2:"numeric", 3:"numeric", 4:"
    numeric"}
20
21
22 # Example 1, a GNB classifier is trained providing "values"
23 print("Results of GNBC:")
24 # create the classifiers
25 c = nb.GaussianNaiveBayes(smooth=0.1, usePrior=True, meta=values)
26 # train the classifier
27 c.fit(data_train, cl_train)
28 # predict
29 p = c.predict(data_test)
30 # evaluation
31 print(c.exactMatch(cl_test, p))
32
33 # ignore the Prior probabilities
34 c.usePrior = False
35 p = c.predict(data_test)
36 print(c.exactMatch(cl_test,p))
37
38
39 # Example 2, a GNB classifier is trained considering all attributes to be
    numeric
40 print("Results of GNBC considering all attributes to be numeric:")
41 # create the classifiers
42 c2 = nb.GaussianNaiveBayes(smooth=0.1, usePrior=True, meta="")
43 # train the classifier
44 c2.fit(data_train, cl_train)
45 # predict
46 p = c2.predict(data_test)
47 # evaluation

```

```

48 print(c2.exactMatch(cl_test, p))
49
50 # ignore the Prior probabilities
51 c2.usePrior = False
52 p = c2.predict(data_test)
53 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:

$$ArgCMax[\log(P(C)) + \log(P(A_1|Pa(A_1), C)) + \dots + \log(P(A_n|Pa(A_n), C))] \quad (2.4)$$

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 6.3.3. 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 $\exp(\text{predict_log_proba})$ 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.

```

1 import numpy as np
2 import PGM_PyLib.augmented as abc
3
4 np.random.seed(0)    # it is not necessary
5 # three classes
6 # 5 attributes
7
8 # 100 instances for training
9 data_train = np.random.randint(0,5,size=(100,5))
10 cl_train = np.random.randint(0,3,size=100)
11 # 50 instances for testing
12 data_test = np.random.randint(0,5,size=(50,5))
13 cl_test = np.random.randint(0,3,size=50)
14
15 # create the classifiers
16 c = abc.augmentedBC(algStructure="auto", smooth=0.1, usePrior=True)
17 # train the classifier
18 c.fit(data_train, cl_train)
19 # predict
20 p = c.predict(data_test)
21 # evaluation
22 print(c.exactMatch(cl_test, p))
23
24 # ignore the Prior probabilities
25 c.usePrior = False
26 p = c.predict(data_test)
27 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* [2]. 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 $\exp(\text{predict_log_proba})$ 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.

```

1 import numpy as np
2 import PGM_PyLib.semiNaive as sn
3
4 np.random.seed(0)      # it is not necessary
5 # three classes
6 # 5 attributes
7
8 # 100 instances for training
9 data_train = np.random.randint(0,5,size=(100,5)).astype(str)
10 cl_train = np.random.randint(0,3,size=100)
11 # 50 instances for testing
12 data_test = np.random.randint(0,5,size=(50,5)).astype(str)
13 cl_test = np.random.randint(0,3,size=50)
14
15 # create the classifiers
16 c = sn.semiNaive(validation=0.8, epsilon=0.01, omega=0.01, smooth=0.1,
17                  nameAtts="auto", usePrior=True)
18 # train the classifier
19 c.fit(data_train, cl_train)
20 # predict
21 p = c.predict(data_test)
22 # evaluation
23 print(c.exactMatch(cl_test, p))
24
25 # ignore the Prior probabilities
26 c.NBC.usePrior = False
27 p = c.predict(data_test)
28 print(c.exactMatch(cl_test,p))
29
30 # show the operations that were applied
31 print(c.opNameAtts)

```

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:

$$\begin{aligned}
 & \text{ArgMax}_{C_1} P(C_1 | \mathbf{Nei}(C_1), \mathbf{X}) \\
 & \text{ArgMax}_{C_2} P(C_2 | \mathbf{Nei}(C_2), \mathbf{X}) \\
 & \dots\dots\dots \\
 & \text{ArgMax}_{C_d} P(C_d | \mathbf{Nei}(C_d), \mathbf{X})
 \end{aligned} \tag{2.5}$$

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 5.1.1 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.

```

1 import numpy as np
2 import PGM_PyLib.BCC as bcc
3 import PGM_PyLib.naiveBayes as nb
4
5 np.random.seed(0)    # it is not necessary
6 # 5 variable classes
7 # three classes for each variable class
8 # 7 attributes
9
10 # 300 instances for training
11 data_train = np.random.randint(0,5,size=(300,7))
12 cl_train = np.random.randint(0,3,size=(300,5))
13 # 100 instances for testing
14 data_test = np.random.randint(0,5,size=(100,7))
15 cl_test = np.random.randint(0,3,size=(100,5))
16
17 # create the classifiers
18 c = bcc.BCC(chainType="parents", baseClassifier=nb.naiveBayes(),
19           structure="auto")
20 # train the classifier
21 c.fit(data_train, cl_train)
22 # predict
23 p = c.predict(data_test)
24 # evaluation
25 print(c.exactMatch(cl_test, p))
26
27 # show the structure
28 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 6.4. 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*, *predict* 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.

```
1 import PGM_PyLib.hierarchicalClassification.BNCC as bncc
2 from sklearn.ensemble import RandomForestClassifier as rfc
3
4 # predict with HCP
5 p1 = bncc.HCP("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
6               arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
7 print(p1)
8
9 #predict with HCA
10 p2 = bncc.HCA("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
11               arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
12 print(p2)
13
14 #predict with HCC
15 p3 = bncc.HCC("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
16               arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
17 print(p3)
18
19 #predict with HBA
20 p4 = bncc.HBA("D_EA_01_FD_b_train_head.arff", "D_EA_01_FD_b_train_data.
21               arff", "D_EA_01_FD_b_test_data.arff", "SP", rfc())
22 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 $[1, \text{length of } O]$.
 - **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.

```

1 import numpy as np
2 import PGM_PyLib.HMM as hmm
3
4 states = ["M1", "M2"]
5 obs = ["H", "T"]
6 PI = np.array( [0.5, 0.5] ) #prior probabilities
7 A = np.array( [[0.5, 0.5], [0.5, 0.5]] ) #transition probabilities
8 B = np.array( [[0.8, 0.2], [0.2, 0.8]] ) #observation probabilities
9
10 # Inializating the model with all its parameters
11 h = hmm.HMM(states=states, observations=obs, prior=PI, transition=A,
12             observation=B)
13
14 O = ["H","H","T","T"] # observation sequence
15
16 # evaluating an observation sequence
17 print("Score of: H,H,T,T")
18 print(h.forward(O))
19
20 # obtaining the most probable state at each time t
21 lmps = [h.MPS(i,O) for i in range(1, len(O)+1) ]
22 print("Most probable state at each time t:")
23 print(lmps)
24
25 # obtaining the most probable sequence of states
26 mpss,score = h.viterbi(O)
27 print("Most probable sequence of states:")
28 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.

```

1 import numpy as np
2 import PGM_PyLib.HMM as hmm
3
4 data=[["H", "H", "T", "H", "T", "H", "T", "H", "T", "T"],
5       ["T", "H", "H", "T", "T"],
6       ["T", "H", "H", "T", "T", "H", "T"],
7       ["H", "T", "T", "T", "T", "H", "T", "T", "T"],
8       ["T", "T", "T", "H", "T", "T"]]
9 h = hmm.HMM() # empty model
10
11 # learning the model from data
12 h.learn(data,tol=0.001,hs=2,max_iter=100,initialization="random",seed=0)
13
14 print("Set of states:")
15 print(h.states)
16 print("Set of observations:")
17 print(h.obs)
18 print("Prior probabilities")
19 print(h.prior)
20 print("Transition probabilities")

```

```
21 print(h.transition)
22 print("Observation probabilities")
23 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, $0 < tempReduction \leq 1$.
 - **optimal : {"MAP","MPM"}, default="MAP"**: Optimal configuration, Maximum A posteriori 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, $0 < tempReduction \leq 1$.

- **optimal** : {"MAP","MPM"}, default="MAP": Optimal configuration, Maximum A posteriori 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.

```

1 import numpy as np
2 from MRF import RMRF as rmrf
3
4 np.random.seed(0) # no mandatory
5
6 s = [i for i in range(6)] #s = "0-6"
7 r = np.random.randint(0,6,size=(5,7)) # RMRF of size 5x7
8 print("initial RMRF\n",r)
9
10 #ICM with MAP
11 mr = rmrf(s,r)
12 print("\nICM, MAP:")
13 r = mr.inference(maxIterations=100, Temp=1.0, tempReduction=1.0, optimal=
    "MAP")
14 print(r)
15 #Metropolis with MAP
16 mr = rmrf(s,r)
17 print("\nMetropolis, MAP:")
18 r = mr.inference(maxIterations=100, Temp=0.01, tempReduction=1.0, optimal=
    "MAP")
19 print(r)
20 #simulated annealing with MPM
21 mr = rmrf(s,r)
22 print("\nSimulated annealing, MPM:")
23 r = mr.inference(maxIterations=100, Temp=0.9, tempReduction=0.8, optimal=
    "MPM")
24 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.

```

1 import numpy as np
2 from MRF import RMRFwO as mrf
3
4 s = [0,1]
5 r = np.zeros((4,4),dtype=int)
6 print("Initial RMRF\n",r)
7 obs=np.array([[0,0,0,0],[0,1,1,0],[0,1,0,0],[0,0,1,0]])
8 print("\nObservation\n",obs)
9
10
11 #ICM with MPM
12 mr = mrf(s,r,obs)
13 print("\nICM, MPM:")
14 r = mr.inference(maxIterations=100, Temp=1.0, tempReduction=1.0, optimal=
    "MPM")
15 print(r)
16 #Metropolis with MPM
17 mr = mrf(s,r,obs)
18 print("\nMetropolis, MPM:")
19 r = mr.inference(maxIterations=100, Temp=0.01, tempReduction=1.0, optimal
    ="MPM")
20 print(r)
21 #simulated annealing with MAP
22 mr = mrf(s,r,obs)
23 print("\nSimulated annealing, MAP:")
24 r = mr.inference(maxIterations=100, Temp=0.9, tempReduction=0.8, optimal=
    "MAP")
25 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) [1,2] 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 6.1) 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.CL_P_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.

```

1 import numpy as np
2 import PGM_PyLib.structures.trees as trees
3
4 np.random.seed(0) # it is not necessary
5
6 # 7 variables
7 # 200 instances
8 data = np.random.randint(0,4,size=(200,3))
9 data = np.concatenate([data, np.random.randint(2,6,size=(200,4))],axis=1)
10
11 # create a instance of CLP_MI
12 clp = trees.CLP_MI(root=0, heuristic=False, smooth=0.1)
13 # create the structure
14 structure = clp.createStructure(data)
15 # show structure
16 print(structure)
17
18 # Use heuristic to automatically select the root of the tree
19 clp.heuristic = True
20 structure = clp.createStructure(data)
21 #show structure
22 print(structure)
23
24 #show the root node of the tree
25 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 5.1.1), 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.

```
1 import numpy as np
2 import PGM_PyLib.structures.trees as trees
3
4 np.random.seed (0)  # it is not necessary
5
6 # 7 variables
7 # 200 instances
8 data = np.random.randint(0,4,size=(200,3))
9 data = np.concatenate([data, np.random.randint(2,6,size=(200,4))],axis=1)
10 #additional variable
11 z = np.random.randint(1,5,size=(200))
12
13 # create a instance of CLP_CMI
14 clp_cmi = trees.CLP_CMI(root=0, heuristic=False, smooth=0.1)
15 # create the structure
16 structure = clp_cmi.createStructure(data, z)
```

```
17 # show structure
18 print(structure)
19
20 # Use heuristic to automatically select the root of the tree
21 clp_cmi.heuristic = True
22 structure = clp_cmi.createStructure(data, z)
23 #show structure
24 print(structure)
25
26 #show the root node of the tree
27 print(clp_cmi.root)
```

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

Chapter 6

Annexes

6.1 Mutual Information

Equation 6.1 was used to estimate the Mutual Information (MI) between two variables.

$$MI(X; Y) = \sum_{y \in Y} \sum_{x \in X} P_{X,Y}(x, y) \log \left(\frac{P_{X,Y}(x, y)}{P_X(x)P_Y(y)} \right) \quad (6.1)$$

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.

6.1.1 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.

6.1.2 Example of Mutual Information

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

```
1 import numpy as np
2 import PGM_PyLib.utils as utils
3
```

```

4 np.random.seed (0) # it is not necessary
5
6 # 200 instances
7 X = np.random.randint(0,4,size=(200))
8 Y = np.random.randint(3,6,size=(200))
9
10 #estimate the MI between two varaibles
11 mi = utils.MI(X, Y, smooth=0.1)
12
13 #show the value obtained
14 print(mi)

```

Listing 6.1: Example Mutual Information

6.2 Conditional Mutual Information

Equation 6.2 was used to estimate the Conditional Mutual Information (CMI) between two variables given a third.

$$CMI(X;Y|Z) = \sum_{z \in Z} \sum_{y \in Y} \sum_{x \in X} P_{X,Y,Z}(x,y,z) \log \left(\frac{P_Z(z) P_{X,Y,Z}(x,y,z)}{P_{X,Z}(x,z) P_{Y,Z}(y,z)} \right) \quad (6.2)$$

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

6.2.1 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.

6.2.2 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.

```

1 import numpy as np
2 import PGM_PyLib.utils as utils
3
4 np.random.seed (0) # it is not necessary
5
6 # 200 instances
7 X = np.random.randint(0,4,size=(200))
8 Y = np.random.randint(3,6,size=(200))
9 #conditional data
10 Z = np.random.randint(10,16,size=(200))
11
12 #estimate the CMI
13 cmi = utils.CMI(X, Y, Z, smooth=0.1)
14
15 #show the value obtained
16 print(cmi)

```

Listing 6.2: Example Conditional Mutual Information

6.3 Estimation of probabilities of $P(A|C_1, C_2, \dots, C_n)$

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

6.3.1 *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, \dots, 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.

6.3.2 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.

```

1 import numpy as np
2 import PGM_PyLib.augmented as pnd
3
4 np.random.seed(0) # it is not necessary
5
6 # 5 variables
7 # variables 0 and 1 can take the values [7,8]
8 # variables 2,3,4 can take the values [10,11,12]
9 # 100 instances
10 data = np.random.randint(7,9,size=(100,3))
11 data = np.concatenate([data, np.random.randint(10,13,size=(100,2))], axis
12                        =1 )
13
14 # variables contains the values that each variable can take
15 variables = {0:[7,8], 1:[7,8], 2:[10,11,12], 3:[10,11,12], 4:[10,11,12]}
16
17 # Example 1: we want to estimate P(0|1,2,3,4)
18 positions = [0,1,2,3,4]
19 cpt = pnd.probsND(variables, positions, smooth=0.1)
20 cpt.estimateProbs(data)
21 #show the conditional probabilities
22 print(cpt.probabilities)
23
24 # Example 2: we want to estimate P(3|1,4)
25 positions = [3,1,4]
26 cpt2 = pnd.probsND(variables, positions, smooth=0.1)
27 cpt2.estimateProbs(data)
28 #show the conditional probabilities
29 print(cpt2.probabilities)
30
31 # The sum of aeA P(a|b,c,...) = 1, forall beB, forall ceC ...
32 #Example 3: below has to sum 1
33 print(cpt2.probabilities[0,0,0] + cpt2.probabilities[1,0,0] + cpt2.
34       probabilities[2,0,0])
35
36 #Example 4: below has to sum 1
37 print(cpt2.probabilities[0,1,1] + cpt2.probabilities[1,1,1] + cpt2.
38       probabilities[2,1,1])

```

Listing 6.3: *exampleCPT.py*: Example of estimation conditional probabilities

Table 6.1 is the conditional probability table (CPT) of $P(V_3|V_1, V_4)$ which was estimated in example 2. We can see that the sum of $P(V_0|V_1 = 7, V_4 = 10)$ is equal to 1

Table 6.1: Example of CPT generated by example 2. $P(V3|V1, V4)$

V1:		7			8		
V4:		10	11	12	10	11	12
V3:	10	0.4426	0.2867	0.3987	0.3005	0.2287	0.1858
	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

(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 6.3.

$$\sum_{a \in A} P(a|c_1, c_2, \dots, c_n) = 1, \forall c_1 \in C_1, \forall c_2 \in C_2, \dots, \forall c_n \in C_n \quad (6.3)$$

6.3.3 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.

6.4 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. 6.1.

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. 6.2.

```
@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 6.1: Example of a header file for hierarchical classification.

```
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 6.2: Example of a data file for hierarchical classification.

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